A Science of Concurrent Programs

Leslie Lamport

version of 2 January 2024

This is a preliminary version of a book. If you have any comments to make or questions to ask about it, please contact me by email. But when you do, include the version date.

I expect there are many minor errors in this version. (I hope there are no major ones.) Anyone who is the first to report any error will be thanked in the book. If you find an error, please include in your email your name as you wish it to appear as well as the version date.

Contents

About This Book (unfinished)

I hope that this book will be published, but that the pdf version will be free and only the printed version will be sold. In the published version, this section will contain practical information about reading the book. Here is some practical information for reading the current version.

- The pdf version contains links to other places in the book. They are colored like this. I have not decided in what color links should be. Besides links to numbered things like equations, there are a few other links in the text from terms to where from they are defined or introduced. I have not decided whether there should be more or none of those other links.

- There will be an index in the published version, but there is none the preliminary version. In the pdf version, page numbers in the index will be uncolored links to those pages.

- The Appendix contains some hopefully interesting digressions and the proofs of most theorems.

- Almost all the examples are tiny, so the reader can concentrate on the principles without being distracted by complications in the examples. These tiny examples have been written in TLA$^+$ and verified by model checking; any errors in what is asserted about them are the result of incorrect transcriptions of what has actually been verified. The TLA$^+$ versions of the examples will be made available on the Web.

- The book contains a few marginal notes. The marginal notes in this version colored like this are notes to myself for things to be checked when preparing the published version.

Acknowledgments (unfinished)

Because the ideas in this book were developed over several decades, it's impossible for me to give credit to everyone who influenced their development. I will therefore restrict this section to acknowledging only people who were coauthors of papers of which I've been an author. They will be listed alphabetically. I haven't decided whether to simply list them, or to indicate what their influence has been. I currently expect there to be between 16 and 24 people listed, but I may change my mind on who might be included.

I will also add acknoledgments to people who read preliminary versions and sent me comments that led to significant changes.

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 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 in which multiple processes, each with its own independent thread of control, are executed concurrently. The first concurrent algorithms were meant to be executed on a single computer, and processes communicated 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*. Scientists are 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.

Engineers are 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 [27, 34]. 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. All other necessary math not introduced along with the science is explained in Chapter 2. Most of that math is taught at universities in an introductory math course for computer science students, though not as rigorously as it is presented here. Scientists should be used to reading math. You may find the math hard if you're an engineer. But unless miseducation 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, 23, 28, 43]. 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 behaviors. I came to realize that correctness can also be expressed by an abstract program—a more abstract, higher-level one than describes 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 [31]. 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. 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 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 “how the program should do it” is the code, but I learned that we need a higher-level, more abstract description of what the code does. To emphasize that programming is more than just coding, I now use the name *coding language* for what are commonly called programming languages. That name is used in this book.

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 to permit efficient execution. I developed abstract programs because I found that the expressiveness of math could make concurrent algorithms even simpler and easier to understand than when written with 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 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$^+$ for writing 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. 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. Model checking is the standard way engineers check those programs.

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 a formula 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 we take it to mean.

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 usually means that there are 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 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.

Is there a citation for this?

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 such programs, although there are simpler sciences that work quite 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

[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$^+$.

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]

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 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 a behavior is called an *invariance*

[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.

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 it has produced the correct output) 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. Since these two versions of the algorithm have the same behaviors, 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. It can be hard or even impossible to write code that allows all those possibilities. Being based on concepts from coding languages, pseudocode also lacks the simplicity and expressiveness of math.

One place we want to allow many possible implementations is in describing what the 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 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 its 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 by 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 [40], Microsoft, and Oracle. Math also provides a new way of thinking about programs that can lead to better programming. There is usually no way to quantify the result of better thinking, but 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, starting with a high-level design written in TLA$^+$. They described their experience in a book [47]. The head of the project, Eric Verhulst, wrote this in an email to me:

> The [TLA$^+$] abstraction helped a lot in coming to a much cleaner architecture. One of the results was that the code size is about 10$\times$ less than in [Virtuoso].[3]

This result was unusual. It was possible only because the design of the

[3]The book states the reduction in code size to be a factor of 5–10. Verhulst explained to me that it was impossible to measure the reduction precisely, so the book gave a conservative estimate; but he believes it was actually a factor of 10.

entire system was described with TLA$^+$. Usually, TLA$^+$ is used to describe only critical aspects of a system that involve concurrency, which represent a small part of the system's code. But this example dramatically illustrates that describing abstract programs with mathematics can produce better programs.

Chapter 2

Ordinary Math

We will write an abstract program as a mathematical formula. The program could be quite complex, leading to a long formula. A long formula with a lot of esoteric mathematics would be impossible to understand. Almost all the math used in our formulas is ordinary math, consisting of arithmetic, simple logic, sets, and functions. You should know it already if you took an introductory university math course for computer science or engineering students. The ordinary math used in this book is explained in this chapter, but from a more sophisticated viewpoint than in introductory math courses. Longer and easier to understand intuitive explanations of most of the math discussed here can be found on the Web.

We will write an abstract program as a TLA formula. While most of that formula consists of ordinary math, TLA is a temporal logic, and temporal logic is not ordinary math. The meaning of TLA formulas will be explained in terms of ordinary math. However, although not hard to understand, temporal logic doesn't satisfy all the properties of ordinary math. Avoiding mistakes when using it requires a good understanding of precisely what its formulas mean. Giving a meaning to the formulas of a logic is called defining a semantics of the logic. This chapter explains how to define a semantics by defining the meaning of the formulas of ordinary math. Even if you're familiar with the math presented here, the explanation of its semantics may be new to you. Therefore, you should at least skim this chapter carefully.

The chapter also describes the method of writing proofs used in this book, which is quite different from how mathematicians write proofs. This method makes the proofs easier to understand, which makes them more reliable. I hope that, some day, engineers will be able to write proofs of correctness of their abstract programs. The proofs mathematicians write are too unreliable to provide confidence in the correctness of programs. Engineers will have to use the proof method presented here, or something like it. Scientists would also benefit from using it.

The description of Euclid's algorithm in Chapter 1 used the program variables x and y. Program variables are different from the variables of ordinary math, such as the ones used in elementary algebra. The values of the program variables x and y in Euclid's algorithm change during execution of the algorithm. The values of the variables x and y of elementary algebra remain the same throughout a calculation. They are like the constants M and N of Euclid's algorithm. For now, we will use the term *variable* to mean a variable of ordinary math, like the ones of elementary algebra.

We begin with what I believe is the hardest math that you will have to know—a branch of math that takes people years to learn. It's much too difficult to explain here, so I will have to assume that you've already learned it. It's called arithmetic.

2.1 Arithmetic and Logic

2.1.1 Numbers

Arithmetic is about numbers. The first numbers you learned about are the positive integers 1, 2, 3, etc. You then learned about more and more kinds of numbers until eventually you learned about the real numbers, which include integers, rational numbers like 3/4, and lots of other numbers like $-\sqrt{2}$ and π (which equals 3.14159...). Although the numbers we actually use are almost always integers, most of our discussion here applies to all real numbers, so we'll let *number* mean *real number.*

We'll use the same notation for the operations of arithmetic that you learned in school—for example $+$, $/$ (division), and $\geq$; except that multiplication is written "$*$" because mathematicians use $\times$ to mean something else. We'll also use one operator of arithmetic that you probably didn't learn as a child. That operator is the modulo operator $\%$, written *mod* by mathematicians. For any integer n greater than 0 and any integer m, the value of $m \% n$ is the remainder when m is divided by n. The precise definition is that $m \% n$ equals the unique integer r satisfying

$$m = d * n + r \quad \text{and} \quad 0 \leq r < n$$

for some integer d.

When studying arithmetic, you probably spent most of your time learning to calculate with numbers. We'll do only a few very simple numerical calculations. What we will use are the properties of numbers that you should have learned—for example:

(2.1) $3 * (\sqrt{3} + \pi) \;=\; (3 * \sqrt{3}) + (3 * \pi)$

2.1.2 Elementary Algebra

After learning about numbers, you probably continued your study of arithmetic with a course in what was called *elementary algebra*, or something similar in another language. That course introduced the concept of variables. A variable represented an unspecified number. You wrote expressions like $x * (y + 3)$ whose value depends on the values of x and y. For example, if x equals 2 and y equals -4, then the value of that expression is $2 * (-4 + 3)$, which equals -2. You learned how to calculate with expressions containing variables. For example, you learned that $x * (y + 3)$ equals $x * y + x * 3$, regardless of the values of x and y.

There are two kinds of expressions in elementary algebra:

- Ones like $x * (y + 3)$, whose value after substituting numbers for the variables is a number. We call such an expression a *numeric* expression. A number is a numeric expression that has no variables.

- Ones like $x * (y + 3) \geq 7$, that are either true or false after substituting numbers for the variables. We call such an expression a *formula*, and we say that the value of a formula after substituting numbers for the variables is either TRUE or FALSE, which are two distinct values called *Booleans*. We often say that a formula is true (or false) for some values of its variables, but that's just a short way of saying that the value of the formula obtained by substituting those values for the variables equals TRUE (or FALSE).

Most of the expressions you wrote in elementary algebra were either formulas or parts of formulas; and most of the formulas you wrote were equations. Most of what you did in elementary algebra consisted of solving equations. That meant finding a single value for each variable such that substituting those values for the variables in the equations made the equations equal TRUE.

We're not interested in solving equations. We will describe programs with formulas, so we need to understand those formulas. This requires understanding some basic concepts of formulas. The formulas of elementary algebra are used to explain these concepts because you're familiar with them.

An important class of formulas are ones that equal TRUE no matter what values are substituted for their variables. Such a formula is said to be *valid*; and the assertion that F is valid is written $\models F$. For example, the truth of formula (2.1) is a special case of:

(2.2) $\models\ p * (q + r) \ = \ p * q + p * r$

The thing we write as $\models F$ is not a formula of elementary algebra. It's an assertion about the formula F. We could call $\models F$ a metamathematical formula or a meta-formula. But $\models F$ is the only kind of meta-formula we need, so we don't need to delve into metamathematics. I will sometimes call $\models F$ a meta-formula to remind you that it's not a formula, but I usually call it something else like an assertion or a condition. When $\models F$ is true for an interesting formula F, mathematicians usually call F a theorem.

Logicians write $\vdash F$ to mean that the formula F can be proved from axioms and proof rules (also called inference rules). They call F a theorem if $\vdash F$ is true. Like most mathematicians, we won't worry about proving formulas from axioms. We prove something by showing that it follows from ordinary math, and we reason about the ordinary math as rigorously as necessary. This book explains how proof rules are used to reduce the proof of correctness of a program to proofs of simpler mathematical properties. Most often used are the rules of ordinary math, which are explained in this chapter. If you need a proof to be completely rigorous, you will have to learn to use a theorem-proving program; you will never prove things from axioms. I will call a formula an axiom when we are assuming it to be true and it can't be deduced from formulas we have already assumed to be true.

We can deduce (2.1) from (2.2) by the following rule:

> **Elementary Algebra Instantiation Rule** Substituting any numeric expressions for (some or all of the) variables in a valid formula yields a valid formula.

For example, applying the rule to (2.2), substituting 3 for p, $\sqrt{3}$ for q, and π for r produces the assertion that (2.1) is a valid formula.

Mathematicians have no standard notation for describing substitution, and the notation I've seen used by computer scientists is impractical for the formulas that arise in describing programs. The notation used in this book, illustrated with substitution for three variables, is that

$$E \text{ WITH } v1 \leftarrow exp1,\ v2 \leftarrow exp2,\ v3 \leftarrow exp3$$

is the expression obtained from the expression E by simultaneously substituting $exp1$ for $v1$, $exp2$ for $v2$, and $exp3$ for $v3$; where $v1$, $v2$, and $v3$ are distinct variables and $exp1$, $exp2$, and $exp3$ are numeric expressions. For example

$$(\, p * (q + r) \text{ WITH } q \leftarrow r,\ r \leftarrow q + s \,) \;=\; p * (r + (q + s))$$

As in this example, the WITH expression is usually enclosed in parentheses when it appears in a formula, otherwise the formula would be difficult to parse.

2.1.3 An Introduction to Mathglish

Math is precise, but this book isn't written in math. It's written in English that explains math. Explaining the precise meaning of math in the imprecise language of English is not easy. To help them do this, English-speaking mathematicians speak and write in a dialect of English I call Mathglish. (I expect mathematicians use similar dialects of other languages.) Mathglish differs from English in two ways: It eliminates some of the imprecision of English by giving a precise meaning to some imprecise English words or to newly invented words, and it makes the written language more compact by using mathematical formulas to replace English phrases.

This book is written in Mathglish. This chapter explains the Mathglish you need to know to read the book. This section discusses the second feature of Mathglish—the use of formulas to replace prose.

Consider these two sentences:

1. Substituting $y + 1$ for z in formula (9.42) yields $x \geq y + 1$.
2. Formula (9.42) shows us that $x \geq y + 1$.

Grammatically, we can see that the two uses of "$x \geq y + 1$" are different. In sentence 1 it's a noun, while in sentence 2 it's a complete clause. In sentence 1, "$x \geq y + 1$" is a formula; in sentence 2 it's an abbreviation for "x is greater than or equal to $y + 1$". This grammatical difference tells us that the two sentences have very different meanings. Sentence 2 asserts that the formula $x \geq y + 1$ is true. The first doesn't tell us whether it is true or false. For example, sentence 1 could be followed by:

Since (9.41) implies $x < y + 1$, this proves that (9.42) is false.

It isn't always possible to tell just from the sentence whether or not it's asserting that a formula is true. For example, we don't know which the following sentence is doing.

3. From (9.42) we deduce $x \geq y + 1$.

In that case, we have to look at the context in which the sentence appears. The formula $x \geq y + 1$ can be true only in a context in which some assumptions have been made about the values of x and $y + 1$—assumptions that are expressed by formulas that are assumed to be true. Sentence 3 asserts that $x \geq y + 1$ is true if and only if formula (9.42) has either been assumed or shown to be implied by assumptions made about x and y. I have tried to make it clear by grammar or context what it means when a formula appears in a sentence in this book.

There's another source of ambiguity in most mathematical writing that I have tried to avoid in this book. Almost no mathematicians other than logicians write the meta-formula $\models F$ differently from the formula F. In most written math, you have to tell from the context which is meant.

2.1.4 Proofs in Elementary Algebra

Engineers now seldom write proofs of correctness of abstract programs. Hand proofs are not reliable, and computer-checked proofs are usually too much work because the proofs must be carried out to a very low level of detail for a computer to check them. However, proof checkers are getting better. Although we don't know when it will happen, machine learning should make it possible for programs to check proofs written at a higher level—proofs that are much easier to write. The proof method described here is the best way I know to write those proofs.

Even before proving correctness becomes easier, engineers as well as computer scientists should learn how to write proofs. Learning how to reason about a class of formulas is part of understanding those formulas. Learning how to prove properties of programs teaches you about programs and their properties.

A mathematician's proof is a sequence of paragraphs written in Mathglish. This way of writing proofs is adequate for very simple proofs. However, it is not reliable for complicated proofs. It is particularly unsuited to handling the many details in a proof of correctness of a program. This book teaches you how to write proofs that are easier to read than the proofs written by mathematicians. Such proofs make it much harder to prove things that aren't true. We will use ordinary paragraph proofs only to prove simple results and to serve as proof sketches that aren't meant to be a proof.

2.1.4.1 An Example

In school, you learned to solve this pair of equations:

(2.3) $3 * x - 2 * y \;=\; 7$ and $7 * x + 3 * y \;=\; 1$

If you can still do it, you will find that the solution is $x = 1$ and $y = -2$. In any case, you can easily calculate that substituting those values for x and y makes both of these equations true. But is that the only solution? Are there other values of x and y for which the equations are true? The procedure you would have followed to find the solution actually proves that it is the only one. As an example, we will write that procedure as a proof.

The method of solving equations (2.3) is based on some rules. One is:

> **Rule EqAdd** ASSUME: $m = n,\; p = q$
> PROVE: $m + p \;=\; n + q$

This rule states that if the values of the variables m, n, p, and q make both the formulas $m = n$ and $p = q$ true, then they make the formula $m + p = n + q$ true. If the values of those variables make $m = n$ or $p = q$ or both of them false, then it tells us nothing about the value of $m + p = n + q$. For example, suppose we know that these two formulas are true:

(2.4) $42 = x + y$ and $2 * x = y + 1$

We can apply the Elementary Algebra Instantiation Rule by substituting $m \leftarrow 42$, $n \leftarrow x + y$, $p \leftarrow 2 * x$, $q \leftarrow y + 1$ in rule EqAdd to deduce this from (2.4):

(2.5) $42 + 2 * x \;=\; (x + y) + (y + 1)$

The second rule used in solving (2.3) is:

> **Rule EqMult** ASSUME: $m = n$
> PROVE: $p * m = p * n$

Another rule that is ubiquitous in mathematics is:

> **Substitution Rule** If $e1$ equals $e2$ and *exp* is an expression containing $e1$ as a subexpression, then *exp* equals the expression obtained from it by replacing $e1$ with $e2$.

For example, I presume you can calculate with elementary algebra expressions, so you know that $42 + 2 * x$ equals $2 * x + 42$ and $(x + y) + (y + 1)$ equals $x + 2 * y + 1$. We can then apply the substitution rule twice, the first substituting (2.5) for *exp* and substituting $42 + 2 * x$ for $e1$ and $2 * x + 42$ for $e2$ to deduce:

$$2 * x + 42 \ = \ (x + y) + (y + 1)$$

A second application of the Substitution Rule, substituting $(x+y)+(y+1)$ for $e1$ and $x + 2 * y + 1$ for $e2$, then implies the truth of:

$$2 * x + 42 \ = \ x + 2 * y + 1$$

We now come to the proof that $x = 1$ and $y = -2$ is the only solution to the equations (2.3). The theorem that asserts this and its proof are in Figure 2.1. The theorem asserts that if the values of x and y make true our two equations, which are numbered as assumptions 1 and 2, then the two equations of the PROVE clause must be true for those values of x and y. This is a precise way to say that 1 and -2 are the only values of x and y that make equations (2.3) true, without having to give *solving* a precise meaning.

The proof consists of a sequence of numbered steps, each but the last one consisting of a formula and a proof. Each of those first seven steps asserts that its formula is true if the theorem's two assumptions are true. That is, each formula is true for all values of x and y for which the assumptions are true. A step's proof explains why the step's formula is true under those assumptions.

I think most people would say that the step 1 formula is obtained by multiplying assumption 1 (our first equation) by 3. The proof gives a more detailed explanation, saying that truth of the formula follows from the truth of assumption 1 and the EqMult rule with 3 substituted for p. I didn't say what expressions were substituted for m and n because I thought it was obvious that the left- and right-hand sides of the equation were substituted for them. But to give you a hint, and in case you've completely forgotten elementary algebra, I reminded you how to multiply $3 * x - 2y$ by 3. (I assumed it was obvious that $3 * 7$ equals 21.)

Step 2 is similar, multiplying the second equation by 2, but the proof leaves the multiplying to you. Step 3 adds the formulas of step 1 and step 2, which means applying Rule EqAdd. I thought it was obvious what was being substituted for the variables in that rule. Steps 4, 5, and 7 are similar, except that, for no particular reason, I used words instead of the symbol $\leftarrow$ to describe the substitutions in the rules. Step 5 is applying the Substitution Rule, with assumption 2 substituted for *exp* and with x and 1 substituted for $e1$ and $e2$. I indicated what was being substituted where, but didn't mention

Theorem ASSUME: 1. $3 * x - 2 * y = 7$,
2. $7 * x + 3 * y = 1$

PROVE: $x = 1$ and $y = -2$

1. $9 * x - 6 * y = 21$

   PROOF: By assumption 1 and Rule EqMult with $p \leftarrow 3$,
   since $3 * (3 * x - 2 * y) = 9 * x - 6 * y$.

2. $14 * x + 6 * y = 2$

   PROOF: By assumption 2 and Rule EqMult with $p \leftarrow 2$.

3. $23 * x = 23$

   PROOF: By steps 1 and 2 and Rule EqAdd.

4. $x = 1$

   PROOF: By step 3 and Rule EqMult, substituting $1/23$ for p.

5. $7 * 1 + 3 * y = 1$

   PROOF: By step 4 and assumption 2 with 1 substituted for x.

6. $3 * y = -6$

   PROOF: By step 5 and Rule EqAdd, substituting -7 for m and n.

7. $y = -2$

   PROOF: By step 6 and Rule EqMult, substituting $1/3$ for p.

8. Q.E.D.

   PROOF: By steps 4 and 7.

Figure 2.1: Proof of uniqueness of the solution to (2.3).

the Substitution Rule. That rule is such a basic part of mathematics that it is taken for granted and never explicitly mentioned in a proof.

Finally, we come to the last statement. "Q.E.D." is an abbreviation for the goal of the proof, which is the PROVE clause of the theorem. In this case, the goal is to prove the two formulas $x = 1$ and $y = -2$. The proof simply points to the steps in which those formulas are proved. A proof always ends with a Q.E.D. step, so we're sure that we've actually proved what we were supposed to.

How we write a proof depends on how hard the proof is and how sophisticated we expect the reader of the proof to be. This proof was written for someone less sophisticated than I expect most readers of this book to be, since I wanted you to concentrate on the proof style rather than on the math. A single-paragraph prose proof would probably be fine for a reader who hasn't forgotten elementary algebra.

2.1.4.2 Longer Proofs

The kind of proof illustrated by Figure 2.1 is more reliable and easier to read than a prose proof. It makes clear the sequence of intermediate results that are being proved and exactly what is being used to prove each of those results. However, a sequence doesn't work for the long proofs needed to prove complex results—such as the correctness of the abstract programs that engineers write.

The method of handling complexity that's obvious to an engineer is hierarchical structuring. Figure 2.2 shows how the proof of Figure 2.1 can be structured. The high-level proof consists of steps 1, 2, and 3. This level-1 proof is suggested by the goal of the theorem, which consists of the two formulas asserted by steps 1 and 2. The proof of its Q.E.D. step (step 3) is the same as the proof of the Q.E.D. step of the Figure 2.1 proof, except the steps it refers to have been renumbered.

The proof of step 1 consists of steps 1.1–1.4. Those steps are the same as steps 1–4 of Figure 2.1, the Q.E.D. of step 1.4 being the goal of that proof, which is $x = 1$. The proofs of steps 1.1–1.4, which have been omitted to save space, are the same as the proofs of the corresponding steps of Figure 2.1. Similarly, the proof of step 2 consists of steps 2.1–2.3, which are the same as steps 5–7 of Figure 2.1.

Note that the proof of step 3 is a prose paragraph, while the proofs of steps 1 and 2 are sequences of steps each with a prose proof. In general, proofs of some or all of those level-2 steps can be further decomposed. Different parts of the proof can be decomposed down to different levels. It's a

1. $x = 1$
   1.1. $9 * x - 6 * y = 21$

   1.2. $14 * x + 6 * y = 2$

   1.3. $23 * x = 23$

   1.4. Q.E.D.
2. $y = -2$
   2.1. $7 * 1 + 3 * y = 1$

   2.2. $3 * y = -6$

   2.3. Q.E.D.
3. Q.E.D.
   PROOF: By steps 1 and 2.

Figure 2.2: Structured version of the proof in Figure 2.1.

good idea to make a Q.E.D. step a simple paragraph that you write first, so you don't waste time proving steps that don't imply the proof's goal.

At the bottom of the hierarchical structure are steps whose proof is written in prose. That prose should be easy to understand, so the reader can be sure that it's correct. How easy that has to be depends on the reader, who may just be you. If you find that the proof isn't easy enough to understand, you should decompose it another level. I've found that the way to avoid errors is to decompose a proof down to the level where the prose proof is obviously correct, and then go one level deeper. For machine-checked proofs, the bottom-level proofs are instructions for the proof checker. If the checker fails to check the proof and you believe the step is correct, then keep decomposing until either the checker says it's correct or you see why it's not.

Long proofs, especially correctness proofs of programs, can be quite deep. For proofs more than three or four levels deep, we use a compact numbering system explained in Section 2.1.10.2 below.

In this example, the theorem's assumptions and goals were mathematical formulas, as were the assertions made by the steps. This should be the case for theorems asserting correctness of programs—except perhaps in some cases at the deepest levels of the proof. In most mathematical proofs, including proofs about the math underlying our science of concurrent programs, the theorem and the assertions of the steps consist of prose statements, such as "x is a prime number." Those statements may be a few sentences long. The prose describes mathematical formulas, but getting the details exactly right isn't as important for those theorems as it is for programs. Hierarchically structured prose proofs are reliable enough for them.

2.1.5 The Semantics of Elementary Algebra

We are now going to over-analyze elementary algebra. We'll perform the kind of hairsplitting that may delight logicians and philosophers, but is of no interest to scientists and engineers who use the math. But in later chapters we will use temporal logic. While not difficult, temporal logic is different from ordinary math in subtle ways. It's easy even for mathematicians to make mistakes when using it. Our over-analysis of elementary algebra will help us avoid those mistakes.

The formulas of elementary algebra are literally strings of characters. To know if how we reason about them makes sense, we need to know what those character strings mean. We must give a semantics to elementary algebra. There's a philosophical question of how we can do that because we have to write the meaning in some language, and how do we know what the words or symbols of that language mean? We're not going to go there.

I believe that I understand arithmetic. I can't tell you what the number 2 or the number 4 is,[1] but I understand that $2 + 2$ equals 4. That kind of understanding is good enough. If I can express the meaning of any formula of elementary algebra in terms of arithmetic, then I am confident that I understand elementary algebra. I assume your knowledge of arithmetic is also good enough.

The meaning of a mathematical theory can be expressed in terms of *collections* and *mappings*. Mathematicians generally call a collection a *set* and call a mapping a *function*. For a particular mathematical theory, such as elementary algebra, we can take collections and mappings to be the same as sets and functions. However, we don't want to restrict the values of variables in abstract programs to be values in some small collection of mathematical theories. We want to allow the value of a variable to be any mathematical object. When we do that, we find that a set is a particular kind of collection, and a function is a particular kind of mapping. Mathematicians sometimes call a collection a *class*.

[1]Mathematicians have defined these numbers, but the explanation of what the things in those definitions are is no better than our explanation of what a number is.

We can't define precisely what a collection is. I could say that a collection is a bunch of things, but I would then have to define what a *bunch* is. I assume you know what a collection is. The things that a collection is a collection of are usually called *values.* For example, the values in the theory of elementary algebra are numbers.

A mapping M from a collection $\mathcal{C}$ of values to a collection $\mathcal{D}$ of values is something that assigns to each value v in $\mathcal{C}$ a value $M(v)$ in $\mathcal{D}$. (The collections $\mathcal{C}$ and $\mathcal{D}$ can be the same.) We say that such a mapping M is a mapping *on* $\mathcal{C}$. For example, we can define a mapping *LengthOfName* from a collection of people to the collection of natural numbers by defining *LengthOfName*(p) to equal the number of letters in the name of p, for every person p in a collection $\mathcal{D}$ of people. If *Jane* is in $\mathcal{D}$, then this defines *LengthOfName*(*Jane*) to equal 4.

A *predicate* is a Boolean-valued mapping—that is a mapping M on a collection $\mathcal{C}$ such that $M(v)$ equals TRUE or FALSE for each value v in $\mathcal{C}$. The meaning of an elementary algebra formula is a predicate on *interpretations*, where an interpretation is a mapping from variables to numbers. We define the meaning $[\![F]\!]$ of a formula F to be the predicate that maps an interpretation to the Boolean value obtained by replacing each variable in F by the value assigned to that variable by the interpretation. For example, if Υ is an interpretation, then $[\![x + y > 42]\!](\Upsilon)$ equals $\Upsilon(x) + \Upsilon(y) > 42$. If $\Upsilon(x) = 3$ and $\Upsilon(y) = 27$, then $[\![x + y > 42]\!](\Upsilon)$ equals $3 + 27 > 42$, which equals FALSE.

If you're not used to reading formulas with Greek letters, go now to Figure 2.3.

This definition of $[\![x + y > 42]\!]$ makes no sense. Here's why. A semantics assigns a meaning to a formula. A formula is a string of characters. Its meaning is a mathematical object. Let's write the string of characters that is the formula in a font like this: `x + y > 42`; and let's write mathematical objects like numbers or variables in the font used throughout this book. I claimed that $[\![$`x + y > 42`$]\!](\Upsilon)$ equals `3 + 27 > 42`, which is a meaningless combination of the numbers 3 and 27 and the four characters `+ > 4 2`.

Elementary algebra has a grammar. The grammar tells us that `x + y` is an expression but `+ x +` isn't, that the `+` in `x + y` is an operator with the subexpressions `x` and `y` as its arguments, and that `x + y * z` is parsed as `x + (y * z)`. I assume you know this grammar, so you understand how the expressions that appear in examples are parsed. To deduce that $[\![$`x+y>42`$]\!](\Upsilon)$ equals $3+27 > 42$ for the particular interpretation Υ, I replaced `x` by $[\![$`x`$]\!](\Upsilon)$, which equals 3, and `y` by $[\![$`y`$]\!](\Upsilon)$, which equals 27. I then replaced the syntactic tokens `+`, `>`, and `42` of elementary algebra by the tokens of arithmetic that are spelled the same. This "punning" works for elementary algebra because it is so closely related to arithmetic. It won't work for TLA, so we use a

I have been told that many engineers freak out when they see a Greek letter like Υ in a formula. If you're one of them, now is the time to get over it. You had no trouble dealing with π as a child; you can now handle a few more Greek letters. They're used sparingly in this book, but sometimes representing a particular kind of object with Greek letters makes the text easier to read. Here are all the Greek letters used in the book, along with their English names. You don't have to remember their names; you just need to distinguish them from one another.

Lowercase

α	alpha	λ	lambda	π	pi	τ	tau
β	beta	μ	mu	ρ	rho	ϕ	phi
δ	delta	ν	nu	σ	sigma	ψ	psi
ϵ	epsilon (also written ε)						

Uppercase

Λ	Lambda	Υ	Upsilon	Π	Pi	Φ	Phi
Δ	Delta						

Figure 2.3: Greek letters used in this book.

more general approach.

We define the meanings of formulas by defining the meanings of all expressions, which for elementary algebra means formulas and numeric expressions. We define rules for operators that can appear in an expression. For example, the rule for the operator symbol `+` in a formula is that for any character strings exp_1 and exp_2 describing numeric expressions and any interpretation Υ, we define $[\![exp_1 \texttt{+} exp_2]\!](\Upsilon)$ to equal $[\![exp_1]\!](\Upsilon) + [\![exp_2]\!](\Upsilon)$. We also define $[\![\texttt{v}]\!](\Upsilon)$ to equal $\Upsilon(v)$ for a variable v; and for any string *dstr* of digits, we define $[\![dstr]\!](\Upsilon)$ to be the number represented by that string of digits. Thus, if $\Upsilon(x) = 3$ and $\Upsilon(y) = 27$, then:

$$\begin{aligned}
[\![\texttt{x + y > 42}]\!](\Upsilon) &= [\![\texttt{x + y}]\!](\Upsilon) > [\![\texttt{42}]\!](\Upsilon) && \text{by the rule for } \texttt{>} \\
&= [\![\texttt{x}]\!](\Upsilon) + [\![\texttt{y}]\!](\Upsilon) > 42 && \text{by the rules for } \texttt{+} \text{ and numbers} \\
&= 3 + 27 > 42 && \text{by the rule for variables} \\
&= \textsc{false}
\end{aligned}$$

It follows immediately from our definition of validity that a formula F is valid if and only if $[\![F]\!](\Upsilon)$ equals TRUE for all interpretations Υ.

If we were going to do all this very rigorously—for example, to write a program to calculate the meanings of formulas—we would define a mapping from formulas to parse trees and apply the rules to the parse tree. But we won't be that compulsive.

Implicit in this exposition is that an interpretation assigns values to all possible variables, not just the ones in any particular expression. The value of $[\![F]\!](\Upsilon)$ for a formula F depends only on the values the interpretation Υ assigns to variables that occur in F. But letting an interpretation assign values to all variables simplifies things, because it means we don't have to keep track of which variables an interpretation is assigning values to.

We assume that there are infinitely many variables. We do this for the same reason we assume there are infinitely many integers even though we only ever use relatively few of them: it makes things simpler not to have to worry about running out of them.

Let's review what we have done. An expression is a string of characters. We define the meaning $[\![\mathit{exp}]\!]$ of an expression *exp* to be a mapping that assigns to every interpretation Υ a value $[\![\mathit{exp}]\!](\Upsilon)$ that is either a number or a Boolean, where an interpretation Υ is a mapping that assigns to each variable v a value $\Upsilon(v)$ that is a number. We define $[\![\mathit{exp}]\!]$ by defining $[\![\mathit{exp}]\!](\Upsilon)$ as follows. An expression is either (i) a variable or (ii) a string of characters that represents an operator *op* applied to its arguments. In case (i), $[\![\mathit{exp}]\!](\Upsilon)$ equals $\Upsilon(\mathit{exp})$. In case (ii), we define for each operator *op* the value of $[\![\mathit{exp}]\!]$ in terms of the values $[\![\mathit{arg}]\!]$ for each argument *arg* of *op* in the expression *exp*. This defines the meaning of the operator *op*. For the operator `+`, we define $[\![\mathit{arg}_1 \texttt{+} \mathit{arg}_2]\!]$ to be the expression $[\![\mathit{arg}_1]\!] + [\![\mathit{arg}_2]\!]$ of arithmetic. The `+` in $[\![\mathit{arg}_1 \texttt{+} \mathit{arg}_2]\!]$ is a one-character string of characters, and the $+$ in $[\![\mathit{arg}_1]\!] + [\![\mathit{arg}_2]\!]$ is an operation of arithmetic. We consider an expression like `42` that represents a number to be an operator that takes no arguments, where $[\![\texttt{42}]\!](\Upsilon)$ equals 42 for every interpretation Υ.

The similarity in the way we write the operator `+` of elementary algebra and the $+$ of arithmetic is obviously not accidental. Syntactically, elementary algebra is an extension of arithmetic to include variables. However, the formula `2 + 3` has a different meaning in elementary algebra than in arithmetic. If we were to give a semantics to a language of arithmetic, then $[\![\texttt{2 + 3}]\!]$ would equal the number 5. In elementary algebra, $[\![\texttt{2 + 3}]\!]$ is a mapping such that $[\![\texttt{2 + 3}]\!](\Upsilon)$ equals 5 for every interpretation Υ. Elementary algebra is an extension of arithmetic in the sense that if exp_1 and exp_2 are two expressions of arithmetic, such as `2 + 3` and `5`, then $[\![\mathit{exp}_1]\!]$ and $[\![\mathit{exp}_2]\!]$ are equal in the semantics of arithmetic iff they're equal in the semantics of elementary algebra.

The purpose of this over-analyzing of elementary algebra was to explain what it means to define the meaning of mathematical operators such as $+$. From now on, we'll be less formal—especially when explaining operators of ordinary math in this chapter.

2.1.6 Arithmetic Logic

In elementary algebra, we can combine numeric expressions using the operators of arithmetic, but we have no operators for combining formulas. What we will call *arithmetic logic* is obtained by adding such operators to elementary algebra. They are the operators for combining the Boolean values TRUE and FALSE. Arithmetic is the study of numbers and the operators on them. So for now, let's call the study of Boolean values and their operators *Boolean arithmetic*. Here are the operators of Boolean arithmetic, and how they are read in Mathglish:

$\neg$	negation	not	$\Rightarrow$	implication	implies
$\wedge$	conjunction	and	$\equiv$	equivalence	if and only if
$\vee$	disjunction	or			

Implication is sometimes written $\rightarrow$ or $\supset$, and $\equiv$ is sometimes written $\Leftrightarrow$ or $\leftrightarrow$.

You've probably already learned about these operators, so they are just defined briefly here as follows. Negation and conjunction are defined by

$$\begin{array}{lcl} \neg\,\text{TRUE} &=& \text{FALSE} \\ \neg\,\text{FALSE} &=& \text{TRUE} \end{array} \qquad \begin{array}{lcl} \text{TRUE} \wedge \text{TRUE} &=& \text{TRUE} \\ \text{TRUE} \wedge \text{FALSE} &=& \text{FALSE} \\ \text{FALSE} \wedge \text{TRUE} &=& \text{FALSE} \\ \text{FALSE} \wedge \text{FALSE} &=& \text{FALSE} \end{array}$$

The other operators can be defined in terms of $\neg$ and $\wedge$ as follows, where $\triangleq$ means *equals by definition*:

$$\begin{array}{lcl} A \vee B &\triangleq& \neg(\neg A \wedge \neg B) \\ A \Rightarrow B &\triangleq& \neg A \vee B \\ A \equiv B &\triangleq& (A \Rightarrow B) \wedge (B \Rightarrow A) \end{array}$$

If you're not familiar with Boolean arithmetic, you should write out the complete definitions of $\vee$, $\Rightarrow$, and $\equiv$, the way it's done above for $\neg$ and $\wedge$. Here is a brief explanation of what these operators and their Mathglish counterparts mean.

$\neg A$ asserts that A is *not* true, where *not* means the same thing in English and Mathglish

$A \land B$ asserts that A is true *and* B is true, where *and* has the same meaning in both languages.

$A \lor B$ asserts that A is true *or* B is true (or both A and B are true). Unlike *or* in English, *or* in Mathglish always allows the possibility that both formulas are true.

$A \Rightarrow B$ asserts that A is true *implies* B is true. This means that B must be true if A is true, but says nothing about B if A is false. Thus, FALSE $\Rightarrow$ TRUE and FALSE $\Rightarrow$ FALSE both equal TRUE. Reading $\Rightarrow$ as *implies* is confusing because A *implies* B in English means that A being true causes B to be true, while *implies* in Mathglish does not. Only in Mathglish would we say that $2 + 2 = 5$ *implies* $2 + 2 = 4$. A good way to understand $\Rightarrow$ and the Mathglish *implies* is that we want

$$(x > 20) \underset{\text{implies}}{\Rightarrow} (x > 10)$$

to be true for all numbers x, and substituting different numbers for x shows that we want $A \Rightarrow B$ to equal TRUE except when $A =$ TRUE and $B =$ FALSE.

$A \equiv B$ asserts that A is true *if and only if* B is true. In other words, $\equiv$ is the equality relation for Boolean values. We read $\equiv$ as *is equivalent to*. Because we often want to express equivalence, written Mathglish has the abbreviation *iff* for *if and only if*. We sometimes write *equals* instead of *is equivalent to* because it's shorter.

Here's how we can write some of the rules and theorems of elementary algebra in arithmetic logic. Rule EqMult asserts that if $m = n$ is true then $p * m = p * n$ is true, for any values of m, n, and p. Therefore, it can be written as

$$\models (m = n) \Rightarrow (p * m = p * n)$$

since $\models$ means true for any values of the variables. Rule EqAdd can be written as

$$\models (m = n) \land (p = q) \Rightarrow (m + p = n + q)$$

We can write the theorem of Figure 2.1 as:

Theorem $(3 * x - 2 * y = 7) \land (7 * x + 3 * y = 1) \Rightarrow (x = 1) \land (y = 2)$

We don't write the $\models$ because it's implied by stating the formula as a theorem. However, we'll see below that if we write the theorem this way, then its proof has to be rewritten.

Observe that $\land$ has higher precedence (binds more tightly) than $\Rightarrow$. The operator $\neg$ has higher precedence than $\land$ and $\lor$, which have higher precedence than $\Rightarrow$ and $\equiv$. Thus

$$\neg A \land B \Rightarrow C \lor D \quad \text{equals} \quad ((\neg A) \land B) \Rightarrow (C \lor D)$$

I don't know how the following expressions should be parsed, so it's best not to write them:

$$A \land B \lor C \qquad A \equiv B \Rightarrow C$$

Boolean operators have lower precedence than other operators, including the operators of arithmetic. The parentheses used above in the statements of the EqMult and EqAdd rules and in the theorem are needed only to make the formulas easier to read.

2.1.7 Propositional Logic

Consider these true assertions about arithmetic logic:

(2.6) (a) $\models (x > 1) \Rightarrow (x > 0)$

(b) $\models (x > 1) \land (y \geq 3) \Rightarrow (x > 1)$

The first expresses a fact about arithmetic. The second tells us nothing about arithmetic. It remains true if we replace $x > 1$ and $y \geq 3$ by any two formulas. It's really a fact about Boolean arithmetic. To study such facts, we temporarily abandon arithmetic logic and introduce a logic of Boolean arithmetic—a logic called *propositional logic*.

A formula of propositional logic is a syntactically correct sequence of variables, Boolean operators, and the Boolean values TRUE and FALSE, where the syntax has been informally described above. Every expression is Boolean-valued, so all the expressions are formulas. The meaning $[\![F]\!]$ of a formula of propositional logic is defined in the same way we defined the meaning of a formula of elementary algebra. We just replace numbers with Boolean values and the operators of arithmetic with the Boolean operators. The meaning of a propositional logic formula is a predicate on interpretations, where an interpretation is a predicate on variables. Thus $[\![\mathtt{A} \land \mathtt{B} \Rightarrow \mathtt{A}]\!](\Upsilon)$ equals $\Upsilon(\mathtt{A}) \land \Upsilon(\mathtt{B}) \Rightarrow \Upsilon(\mathtt{A})$, where $\land$ and $\Rightarrow$ in the formula are characters, while those same symbols in the meaning are the corresponding operators of Boolean arithmetic. As in arithmetic logic, $\models A$ for a propositional logic formula A asserts that $[\![A]\!](\Upsilon)$ equals TRUE for all interpretations Υ.

A valid formula of propositional logic is often called a *tautology.* Here are some tautologies that are easy to check by substituting the two Booleans for A; but you should find them obvious from the explanation of the operators given above. (Remember that $\equiv$ is the equality operator of Boolean arithmetic.)

$$\begin{array}{lll}
\models \neg\neg A \;\equiv\; A & \models \text{TRUE} \wedge A \;\equiv\; A & \models \text{FALSE} \wedge A \;\equiv\; \text{FALSE} \\
\models A \vee \neg A \;\equiv\; \text{TRUE} & \models A \wedge \neg A \;\equiv\; \text{FALSE} & \models (\text{TRUE} \Rightarrow A) \;\equiv\; A \\
\models A \wedge A \;\equiv\; A & \models A \vee A \;\equiv\; A & \models (\text{FALSE} \Rightarrow A) \;\equiv\; \text{TRUE}
\end{array}$$

The following six tautologies have names. You should find the first five easy to remember because they can be obtained from familiar laws of arithmetic by substituting $\wedge$, $\vee$, and $\equiv$ for $*$, $+$, and $=$. The sixth is a bonus that has no counterpart in ordinary arithmetic.

$$\begin{array}{ll}
\text{Commutativity:} & \models A \wedge B \;\equiv\; B \wedge A \\
 & \models A \vee B \;\equiv\; B \vee A \\
\text{Associativity:} & \models (A \wedge B) \wedge C \;\equiv\; A \wedge (B \wedge C) \\
 & \models (A \vee B) \vee C \;\equiv\; A \vee (B \vee C) \\
\text{Distributivity:} & \models A \wedge (B \vee C) \;\equiv\; (A \wedge B) \vee (A \wedge C) \\
 & \models A \vee (B \wedge C) \;\equiv\; (A \vee B) \wedge (A \vee C)
\end{array}$$

Just as associativity of $*$ implies that we can write $a * b * c * d$ without parentheses because it doesn't matter in which order we compute the products, associativity of $\wedge$ implies that we can write $A \wedge B \wedge C \wedge D$. Similarly, associativity of $\vee$ means we can write $A \vee B \vee C \vee D$ without parentheses. And as in arithmetic, commutativity implies we can write conjunctions or disjunctions in any order.

The following two tautologies, which are called De Morgan's laws, show how to move negation over disjunction and conjunction when calculating with Boolean arithmetic.

$$\models \neg(A \wedge B) \;\equiv\; \neg A \vee \neg B \qquad \models \neg(A \vee B) \;\equiv\; \neg A \wedge \neg B$$

They are important for understanding and manipulating formulas, and you should internalize them—for example by understanding why:

$$\neg((x = 1) \wedge (y = 2)) \;\text{ is equivalent to }\; (x \neq 1) \vee (y \neq 2)$$

An important property of implication is:

Transitivity: $\models (A \Rightarrow B) \wedge (B \Rightarrow C) \;\Rightarrow\; (A \Rightarrow C)$

If $A \Rightarrow B$ is true (in some context), then we say that A is *stronger* than B, or B is *weaker* than A. That's a Mathglish abbreviation of "stronger/weaker than *or equivalent to*", since $A \equiv B$ implies $A \Rightarrow B$ and $B \Rightarrow A$. Stronger means implying at least as many formulas because, by transitivity, A stronger than B means that if $B \Rightarrow C$ is true then so is $A \Rightarrow C$.

We usually manipulate propositional logic formulas to see if one formula implies or is equivalent to another. Besides the tautologies given above, all you need to know to do that are the definition of implication and that $\equiv$ satisfies the usual properties of equality. When manipulating a formula, if you can't use transitivity of $\Rightarrow$, it's usually best to expand the definition of $\Rightarrow$ so the formula contains only the operators $\neg$, $\wedge$, and $\vee$.

When we describe programs with formulas, the operators of propositional logic are used more often in those formulas than any other mathematical operators, except perhaps "$=$". It's therefore important to understand them well and to be comfortable using propositional logic.

Fortunately, propositional logic is really simple because it's based on Boolean arithmetic; and Boolean arithmetic is as simple as math can get because it's the arithmetic of just two values. However, you may not be comfortable using propositional logic, either because you never learned it or because you haven't spent as much time learning it as you spent learning elementary algebra. In that case, you may want to supplement what you learn here by reading about it on the Web. Propositional logic is also called Boolean algebra, and TRUE and FALSE are also sometimes written as $\top$ and $\bot$ or as 1 and 0.

Even though propositional logic is simple, long formulas can be difficult to understand. I sometimes find it hard to see if two propositional logic formulas are equivalent. Instead of spending time trying to use tautologies to calculate if they're the same, I ask a computer. There are propositional logic calculators on the Web that can be used for that. They can also be helpful learning tools.

2.1.8 The Propositional Logic of Arithmetic

2.1.8.1 The Logic

The meanings of $[\![F]\!]$ and $\models F$ depend on the logic we're talking about. So far, we've used it for formulas of arithmetic logic and of propositional logic. There's an important relation between those two logics: if F is a valid formula of propositional logic and we substitute formulas of arithmetic logic for the variables of F, then we obtain a valid formula of arithmetic logic. For example, the assertion (2.6b) is obtained by the obvious substitutions in the tautology $\models A \wedge B \Rightarrow A$. This relation between propositional logic and the logic of arithmetic underlies much of the reasoning in proofs about elementary algebra formulas.

Instead of having two separate logics, we modify arithmetic logic so it extends propositional logic in much the way that elementary algebra extends arithmetic. We do this by adding Boolean-valued variables to arithmetic logic. This means that we have two different kinds of variables: numeric-valued and Boolean-valued. We'll eliminate the "valued" and call then numeric and Boolean variables. We call the resulting logic the *propositional logic of arithmetic*.

In this logic, an interpretation assigns numbers to numeric variables and Booleans to Boolean variables. I have adopted the convention of using lower-case letters for numeric variables and upper-case ones for Boolean variables. But this is just a convention, and in principle there are two different variables that we call x. We can assume that those two variables are distinguishable in some way—perhaps by being printed in different colors. Whenever we use a variable it will be clear in which color it should be printed, so we don't bother coloring it. And of course, we'll never use the same name with both colors in the same formula. Later in this chapter, we'll see that we don't need these two different kinds of variables. But in Chapter 3, when we start using temporal logic, we'll again need two different kinds of variables.

Having incorporated propositional logic into arithmetic logic, we don't have to "import" tautologies from propositional logic to reason about elementary algebra. Those tautologies have become tautologies of the proposition logic of arithmetic. To use them, we extend the Elementary Algebra Instantiation Rule to:

> **Propositional Logic of Arithmetic Instantiation Rule** Substituting any numeric expressions for (some or all) numeric variables and any formulas for (some or all) Boolean variables in a valid formula yields a valid formula.

One propositional logic tautology often used in proofs is the transitivity of $\Rightarrow$. Just as we write $x < y < z$ as an abbreviation for $(x < y) \wedge (y < z)$, we sometimes write $A \Rightarrow B \Rightarrow C$ for $(A \Rightarrow B) \wedge (B \Rightarrow C)$. A nice way to write a proof of $A \Rightarrow Q$ is:

$A \Rightarrow B$	Proof of $A \Rightarrow B$.
$\Rightarrow C$	Proof of $B \Rightarrow C$.

$$\begin{array}{ll} \vdots & \\ \Rightarrow Q & \text{Proof of } P \Rightarrow Q. \end{array}$$

This works well if the proof of each implication is short. It's my favorite way of writing a lowest-level prose proof of a hierarchically structured proof.

Another useful tautology is

$$\models (P \Rightarrow Q) \equiv (P \land \neg Q \Rightarrow \text{FALSE})$$

This is the basis of a proof by contradiction, in which we prove that P implies Q by assuming P and $\neg Q$ and obtaining a contradiction (which implies FALSE). Many mathematicians dislike proofs by contradiction, which they find inelegant. If you write a proof to make sure that what you're trying to prove is true, then you should always write a proof by contradiction. It's never harder and can make it easier to write the proof. I like to view proofs by contradiction in terms of this tautology:

$$\models (P \Rightarrow Q) \equiv (P \land \neg Q \Rightarrow Q)$$

It shows that to prove $P \Rightarrow Q$, we can assume not just P but also $\neg Q$ when proving Q. This gives us an additional hypothesis. Moreover, it's a very strong hypothesis. If $P \Rightarrow Q$ is true, then P implies that $\neg Q$ is equivalent to FALSE, which is the strongest possible hypothesis (since FALSE implies anything). If you wind up not using the additional hypothesis, you can just delete it.

2.1.8.2 More About Proofs

Consider the theorem of Figure 2.1 of Section 2.1.4.1. As we saw in Section 2.1.6, it makes an assertion of the form $\models T$, for a formula T of arithmetic logic containing the variables x and y. To prove it, we had to prove that T is true when any numbers are substituted for x and y. We do this by assuming that x and y equal some unspecified numbers and showing that T is true for those unspecified numbers. Expressed semantically, $\models T$ asserts that $[\![T]\!](\Upsilon)$ is true for all interpretations Υ. We prove this by proving $[\![T]\!](\Upsilon)$ is true for some particular unspecified interpretation Υ. The goal of the proof of

Theorem T

is to show that $[\![T]\!](\Upsilon)$ is true for the unspecified interpretation Υ.

The formula T in our example is of the form $A \Rightarrow P$ for formulas A and P. Most of the theorems we prove have this form. The assertion $\models A \Rightarrow P$

means that $[\![A \Rightarrow P]\!](\Upsilon)$ is true for every interpretation Υ, which is true iff $[\![P]\!](\Upsilon)$ is true for every interpretation Υ for which $[\![A]\!](\Upsilon)$ is true. In other words, we can prove $\models A \Rightarrow P$ by assuming that Υ is an arbitrary interpretation such that $[\![A]\!](\Upsilon)$ is true and proving $[\![P]\!](\Upsilon)$ is true for that interpretation. Writing

Remember that iff means *if and only if.*

Theorem ASSUME: A
PROVE: P

asserts that $\models A \Rightarrow P$ is true, but the goal of the proof is to show that $[\![P]\!](\Upsilon)$ is true for an interpretation Υ, assuming $[\![A]\!](\Upsilon)$ is true for that interpretation. In other words, the goal of the proof is to show that P is true when we can assume that A is true throughout the proof. Therefore, the Q.E.D. step asserts that P is true (which completes the proof of $A \Rightarrow P$), not that $A \Rightarrow P$ is true.

In plain Mathglish, if the theorem asserts $A \Rightarrow P$, then the goal of the proof is to prove $A \Rightarrow P$, with no additional assumption. If we write the theorem as an ASSUME/PROVE, then the goal of the proof is to prove P, using the assumption that A is true. Either way, we're proving the same thing: $\models A \Rightarrow P$.

In our example, formula A equals $B \wedge C$. Assuming that $B \wedge C$ is true is the same as assuming that B is true and C is true. Writing $B \wedge C$ as two separate assumptions allows us to give them each a number, so we can indicate in the proof which of the two conjuncts is being used in proving an individual step. This makes the proof easier to read. The goal P is also a conjunction, but there is seldom any reason to number the individual conjuncts of a goal.

The ASSUME/PROVE construct is not limited to the statement of the theorem. It can be used as any step of a proof. The formulas in the ASSUME clause as well as any assumptions in effect for that statement can be assumed in the statement's proof.

A formula P is often proved by showing that to prove P it suffices to prove some other formula Q, and then to prove Q. If P is a statement in a hierarchically structured proof, this proof can be written as:

2.3. P
  2.3.1. $Q \Rightarrow P$
    Proof of $Q \Rightarrow P$
  2.3.2. Q
    Proof of Q

2.3.3. Q.E.D.
  PROOF: By steps 2.3.1 and 2.3.2.

The problem with this structure is that the proof of Q, which is likely to be the main part of the proof of P, is one level deeper than the proof of P. (It starts with statement 2.3.2.1.) That extra level of depth serves no purpose and makes the proof harder to read. Instead, we write the proof of P like this:

2.3. P
  2.3.1. SUFFICES: Q
    Proof of P, assuming that Q is true.
  2.3.2. ...
    ⋮
  2.3.7. Q.E.D.
    Proof that steps 2.3.2–2.3.6 prove Q.

Starting with step 2.3.2, the SUFFICES statement changes the goal of the proof of step 2.3 from P to Q. The proof of step 2.3.1 has P as its goal and Q as an additional assumption that can be used. In other words, the proof of 2.3.1 is the same as if the statement were:

2.3.1. ASSUME: Q
  PROVE: P

The SUFFICES construct can be used with ASSUME/PROVE too, as in:

2.3. P
  2.3.1. SUFFICES: ASSUME: A
    PROVE: Q
    Proof of P, assuming that $A \Rightarrow Q$ is true.

In addition to changing the goal of the proof of step 2.3 from P to Q, this SUFFICES statement also adds A to the current assumptions of that proof. The proof of 2.3.1 is then the same as the proof of:

2.3.1. ASSUME: $A \Rightarrow Q$
  PROVE: P

If A is a conjunction, then its conjuncts can be listed and given numbers in the ASSUME clause of a SUFFICES statement, the same as in an ordinary ASSUME/PROVE statement.

A common proof strategy is to decompose a proof into cases. For example, the proof that each step of Euclid's algorithm maintains the truth of $GCD(x, y) = GCD(M, N)$ might be split into the three cases $x > y$, $y > x$, and $x = y$. This is done with the CASE statement. If G is the current goal of the proof, then the statement CASE A is an abbreviation of:

ASSUME: A PROVE: G

In a proof by case splitting, a sequence of CASE steps is followed by a Q.E.D. step whose proof shows that the cases cover all possibilities.

We have now seen all the kinds of statements needed to write a proof, except for one: a statement for making definitions local to the proof. It's described in Section 2.4.3.1.

2.1.9 Predicate Logic

Predicate logic is an extension of propositional logic. It includes the operators of propositional logic, and all propositional logic tautologies are valid formulas of predicate logic. It extends propositional logic because the value of a variable is not either TRUE or FALSE, but instead is a predicate on some collection of values—the same collection for all variables. (Propositional logic is equivalent to the special case in which that collection contains just a single value.)

Predicate logic is normally described as adding to propositional logic two additional operators called *quantifiers*. There is a third mathematical operator that seems to be considered an add-on to predicate logic, if it is considered at all, but that I consider part of the logic. It's described after the descriptions of the quantifiers.

We won't bother studying predicate logic in general, just the special case in which predicates are predicates on the set $\mathbb{R}$ of real numbers. This means that we will consider the logic obtained by adding the operators of predicate logic to arithmetic logic—a logic we will call the *predicate logic of arithmetic*. All the properties of this logic that don't depend on the laws of arithmetic are true of predicate logic in general.

2.1.9.1 Quantifiers

The symbols, names, and Mathglish pronunciations of quantifiers are:

$\forall$	universal quantification	*for all*
$\exists$	existential quantification	*there exists*

They have the following meanings, where v is any numeric variable and F is any formula of the predicate logic of arithmetic:

$\forall\, v : F$ is true iff F is true when any number is substituted for v.

$\exists\, v : F$ is true iff there is some number that, when substituted for v, makes F true.

To be a bit more precise, for any interpretation Υ, let

$$\Upsilon \text{ EXCEPT } v \leftarrow r$$

be the mapping that's the same as Υ except it assigns to the variable v the number r. For any interpretation Υ:

$[\![\forall\, v : F]\!](\Upsilon)$ equals TRUE iff $[\![F]\!](\Upsilon \text{ EXCEPT } v \leftarrow r)$ equals TRUE for every number r.

$[\![\exists\, v : F]\!](\Upsilon)$ equals TRUE iff $[\![F]\!](\Upsilon \text{ EXCEPT } v \leftarrow r)$ equals TRUE for some number r.

When parsing a formula, the scope of a quantifier extends as far as possible—for example, until terminated by the end of the formula or by a right parenthesis whose matching left parenthesis precedes the quantifier.

The following formula is an example of universal quantification:

(2.7) $\forall\, x \;:\; y * x^2 \geq x^2$

Since (i) $x^2 \geq 0$ for any number x and (ii) $y * r \geq r$ for all $r \geq 0$ iff $y \geq 1$, this formula equals TRUE iff $y \geq 1$. Thus, (2.7) is equivalent to $y \geq 1$.

The following formula asserts that there exists a (real) number whose square equals y:

(2.8) $\exists\, x \;:\; y = x^2$

Since a real number y has a square root (that's a real number) iff $y \geq 0$, this formula is equivalent to $y \geq 0$.

The two quantifiers are related by these theorems:

(2.9) $\models (\forall\, v : F) \;\equiv\; (\neg \exists\, v : \neg F) \qquad \models (\exists\, v : F) \;\equiv\; (\neg \forall\, v : \neg F)$

You should be able to check that they follow from the informal definitions of $\forall$ and $\exists$. We can take either of these theorems to be the definition of one of the quantifiers in terms of the other.

I like to consider $\forall\, v$ and $\exists\, v$ to be the operators, and to consider $\forall$ and $\exists$ (and the colon ":") to be pieces of syntax. These two operators are *dual* to each other, meaning that to negate the application of either of them to a formula F, we replace the operator by its dual and replace F by $\neg F$. This duality is expressed by these theorems:

$$(2.10) \quad \models \neg(\forall\, v : F) \equiv (\exists\, v \,:\, \neg F) \qquad \models \neg(\exists\, v : F) \equiv (\forall\, v \,:\, \neg F)$$

They follow from (2.9) by negating both sides of its equivalence relations.

In formulas (2.7) and (2.8), x is called a *bound* variable. The variable x doesn't really occur in those predicates. If we replace x by another variable z in (2.8), we get $\exists\, z : z^2 = y$ — a formula that is not just equivalent to (2.8), but is really just a different syntax for the same formula. When we call x a bound variable of a formula, we are making a statement about syntax, regarding the formula as a string of characters. The variables that really do occur in a formula, like the variable y in (2.8), are called *free* variables and are said to *occur free* in the formula.

We can write $\forall\, v$ or $\exists\, v$ in a context in which v already has a meaning—for example, in a formula of the following form, where F, G, and H are formulas:

$$(2.11) \ \forall\, v \,:\, (F \land (\exists\, v \,:\, G) \land H)$$

An occurrence of v in formula F or formula H and an occurrence of v in formula G mean different things. We can say that the meaning of v is changed within the scope of the $\exists\, v$ operator. However, it's easier to think of there being two different variables that, for convenience, we write as "v". Imagine that every time we introduce a variable that's written as "v", we're actually introducing a brand-new variable that we've never used before. If we've already used 142 different variables written as "v", then (2.11) should really be written:

$$\forall\, v_{143} \,:\, (F \land (\exists\, v_{144} \,:\, G) \land H)$$

But we're lazy, so we abbreviate it as (2.11). We should never explicitly write a formula such as (2.11), but we'll see that such formulas can arise implicitly. So, we have to remember that those unwritten subscripts are there. (In fact, when a program parses (2.11), it would probably represent v_{143} and v_{144} by two different *variable* objects that have a *name* field with value "v".)

2.1.9.2 A Subtlety Explained

The theorems in (2.9) and (2.10) are different from propositional logic tautologies in a subtle way. We apply a tautology like $\models (F \land G) \equiv (G \land F)$

by substituting formulas for F and G to get a theorem. We can't substitute a formula for F in (2.9) or (2.10) to get a useful theorem. As an example, suppose we want to substitute $v > y$ for F in the first theorem of (2.9). That theorem is really something like:

$$(2.12) \quad \models\ (\forall\, v_{13} : F) \ \equiv\ (\neg\, \exists\, v_{14} : \neg F)$$

The v in the formula $v > y$ also has some subscript, depending on where that formula comes from—perhaps it's v_{64}. Then the substitution produces something like:

$$(2.13) \quad \models\ (\forall\, v_{13} : v_{64} > y_{37}) \ \equiv\ (\neg\, \exists\, v_{14} : \neg(v_{64} > y_{37}))$$

If a variable x does not appear in a formula G, then $\forall\, x : G$ and $\exists\, x : G$ are both equivalent to G. So, (2.13) is equivalent to

$$\models (v_{64} > y_{37}) \ \equiv\ \neg\neg(v_{64} > y_{37})$$

This is valid, but it's not the theorem we wanted.

One possible approach is to say that when substituting for F in the scope of a quantifier like $\forall\, v_{13}$, any variable named v is replaced by v_{13}. Substituting $v_{62} > y_{37}$ for F in (2.12) then yields

$$\models\ (\forall\, v_{13} : v_{13} > y_{37}) \ \equiv\ (\neg\, \exists\, v_{14} : \neg(v_{14} > y_{37}))$$

which is the theorem we want. However, we'll see in Section 2.4.1.1 that this kind of substitution is not sound, allowing the deduction of false theorems.

The correct approach is that we don't just substitute for F when it appears in theorems like (2.9), but for the bound variable v as well. In addition to substituting the formula $v_{64} > y_{37}$ for F, we substitute the variables v_{64} for both v_{13} and v_{14} to get

$$(2.14) \quad \models\ (\forall\, v_{64} : v_{64} > y_{37}) \ \equiv\ (\neg\, \exists\, v_{64} : \neg(v_{64} > y_{37}))$$

Of course, this is also what we want to do if v_{64} were x_{46} or any other variable.

This all works, and it's quite natural to substitute for the variable v when applying theorems (2.9), since it's unlikely that the we will want to apply them to formulas whose bound variable happens to be named v. But I find it somewhat inelegant, and inelegance bothers me because I worry that it might indicate a potential problem. Fortunately, there's another way of viewing quantification which I find more elegant that justifies what we're doing.

The alternative view is to interpret $\forall\, v : F$ with its formula F as a formula $\forall\,(F)$ with no bound variable, where F is not a formula but a mapping. The formula $\forall\, v : v > y$ is viewed as the formula $\forall\,(M)$, where M is the mapping defined by $M(v) \triangleq v > y$. If we write this mapping M as $v \mapsto v > y$, then we see that $\forall\, v : v > y$ is way of writing $\forall\,(v \mapsto v > y)$. The theorem (2.12) becomes $\models \forall\,(F) \equiv \neg\exists\,(\neg F)$, where F is a mapping and $\neg F$ is the mapping defined by $(\neg F)(v) = \neg F(v)$ for all values v. Substituting $v_{64} \mapsto v_{64} > y_{37}$ for F in $\models \forall\,(F) \equiv \neg\exists\,(\neg F)$ yields:

$$\models\ \forall\,(v_{64} \mapsto v_{64} > y_{37})\ \equiv\ \neg\,\exists\,(\neg(v_{64} \mapsto (v_{64} > y_{37})))$$

Using the definition of $\neg F$ for a mapping F to write $\neg(v_{64} \mapsto \ldots)$ as $v_{64} \mapsto \neg \ldots$, this theorem is what we usually write as (2.14). This view shows that replacing the bound variables v of (2.12) by the v in the formula that is substituted for F is the natural thing to do.

We will use the standard notation for quantifiers. However, I find that this alternative view helps us understand what we're doing.

2.1.9.3 Predicate Logic Reasoning

Reasoning about formulas of the predicate logic of arithmetic requires rules of predicate logic. The predicate logic tautologies (2.9) and (2.10) provide such rules. Two more tautologies are

$$\begin{aligned} (2.15)\quad &\models (\forall\, v \,:\, F \land G)\ \equiv\ ((\forall\, v \,:\, F)\ \land\ (\forall\, v \,:\, G)) \\ &\models (\exists\, v \,:\, F \lor G)\ \equiv\ ((\exists\, v \,:\, F)\ \lor\ (\exists\, v \,:\, G)) \end{aligned}$$

You should be able to find examples to show that these assertions become false if $\forall$ and $\exists$ are interchanged. Another simple but useful rule is that if the variable v does not occur free in formula F, then $\forall\, v : F$ and $\exists\, v : F$ are both equivalent to F. For example, if v does not occur free in F, then the first theorem of (2.15) implies that $\forall\, v : (F \land G)$ is equivalent to $F \land (\forall\, v : G)$.

There are four more rules that are often used. Two of them are for proving the two kinds of quantified formulas, the other two are for using each kind of quantified formula to prove something else. The first two are called quantifier introduction rules, the second two are called quantifier elimination rules. The first uses a new kind of assumption in a proof. The others are simple implications, which can be proved with an ASSUME/PROVE as described in Section 2.1.8.2.

$\forall$ **Introduction** To prove $\forall\, v : F$, we need to show that F is true for any value of v. We do that by proving F under the assumption that v is a brand new variable about which we know nothing. The following ASSUME/PROVE in a theorem or in a proof statement is equivalent to the formula $\forall\, v : F$.

ASSUME: NEW v
PROVE: F

However, its proof has F as its goal, and the variable v in F and in any formulas in the proof is regarded as a new variable, unrelated to any occurrence of v in formulas outside the proof. We could write a proof of $\forall\, v : F$ like this:

```
3.2. $\forall\, v \ : \ F$
  3.2.1. SUFFICES: ASSUME: NEW $v$
                   PROVE:  $F$
    PROOF: Obvious (because the ASSUME/PROVE asserts $\forall\, v : F$).
   ⋮
  3.2.7. Q.E.D.
    Proof that steps 3.2.2–3.2.6 imply $F$.
```

We could also eliminate one level of proof and the SUFFICES step by having step 3.2 simply assert the ASSUME/PROVE.

$\exists$ **Introduction** To prove $\exists\, v : F$, we have to show that there is a value of v that makes F true. We do that by explicitly describing that value. That is what this tautology asserts, where *exp* is a numeric expression:

$$\models (F \text{ WITH } v \leftarrow \mathit{exp}) \ \Rightarrow\ \exists\, v \ : \ F$$

$\forall$ **Elimination** The formula $\forall\, v : F$ asserts that F is true for all values of v. We deduce from this formula that F is true when we substitute a particular expression for v. That is asserted for the numeric expression *exp* by:

$$\models (\forall\, v \ : \ F) \ \Rightarrow\ (F \text{ WITH } v \leftarrow \mathit{exp})$$

We then use the formula F WITH $v \leftarrow \mathit{exp}$ to prove our goal.

$\exists$ **Elimination** Suppose $\exists\, v : F$ is true and $F \Rightarrow G$ is true when v has any value. This implies that G is true for the particular value of v that makes F true, so $\exists\, v : G$ is true. Thus, we have the following rule:

$$\models F \Rightarrow G \quad \text{implies} \quad \models (\exists\, v : F) \Rightarrow (\exists\, v : G)$$

This doesn't look like an $\exists$ elimination rule because we use $\exists\, v : F$ to prove another existentially quantified formula $\exists\, v : G$, so we haven't eliminated the $\exists$. It becomes an elimination rule if v does not occur free in G, so $\exists\, v : G$ equals G. (The rule is usually stated with $\exists\, v : G$ replaced by G and the side condition that v does not occur free in G.)

We must be careful when using the WITH constructs in these rules. If F is the formula $(v > 0) \Rightarrow \exists\, v : G$, then F WITH $v \leftarrow \mathit{exp}$ equals $(\mathit{exp} > 0) \Rightarrow \exists\, v : G$. No substitution is performed in G because, within G, the bound variable v is not the same variable as the v in $v > 0$.

2.1.9.4 The CHOOSE Operator

Mathematicians often define something in terms of its properties. For example, they might define $\sqrt{r}$ for a real number r to be the positive real number such that $(\sqrt{r})^2 = r$. We can express such a definition using an operator invented by the mathematician David Hilbert in the 1920s. I didn't learn about this operator until about 25 years after I completed my studies; I suspect it's still unknown to most mathematicians. Hilbert called it ε, but I think it's better to call it CHOOSE. We can use this operator to define the square root as follows (since we are assuming all predicates are predicates on $\mathbb{R}$):

$$\sqrt{r} \;\triangleq\; \text{CHOOSE } s : (s \geq 0) \land (s^2 = r)$$

In general, the expression CHOOSE $v : P$ equals a value e that makes P true when e is substituted for v. If there is no such e, then the value of the expression is unspecified. If there is more than one such value e, then the expression can equal any one of those values. For example, define the mapping *ASqrt* by:

$$\mathit{ASqrt}(r) \;\triangleq\; \text{CHOOSE } s : s^2 = r$$

Then $\mathit{ASqrt}(4)$ might equal 2 and $\mathit{ASqrt}(9)$ might equal -3. Since this is math, $\models \mathit{ASqrt}(4) = \mathit{ASqrt}(4)$ is true. The value of $\mathit{ASqrt}(4)$ may be 2 or -2. But whichever value it equals, like every mathematical expression with no free variable, it always equals the same value.

Formally, CHOOSE is defined by the following axioms:

(2.16) (a) $\models (\exists\, v : P) \;\Rightarrow\; (P \text{ WITH } v \leftarrow (\text{CHOOSE } v : P))$
(b) $\models (\forall\, v : P \equiv Q) \;\Rightarrow\; ((\text{CHOOSE } v : P) = (\text{CHOOSE } v : Q))$

If there is more than one value of x for which P equals TRUE, then CHOOSE $x : P$ can equal any of those values. But it always equals the same value.

No matter how often I repeat that the CHOOSE operator always chooses the same value, there are engineers who think that CHOOSE is nondeterministic, and they try to use it to describe nondeterminism in a program. I've also heard computer scientists talk about "nondeterministic functions".[2] There's no such thing. There's no nondeterminism in mathematics. Nondeterminism is important in concurrent programs, and it's easy to describe mathematically. Adding nondeterminism to math for describing nondeterminism makes as much sense as adding water to math for describing fluid dynamics.

An expression CHOOSE $v : P$ is most often used when there is only a single choice of v that makes P true, as in the definition of $\sqrt{r}$ above. Sometimes, it appears within an expression whose value doesn't depend on which value of v satisfying P is chosen.

2.1.10 More About Proofs

2.1.10.1 Assume/Prove in General

We have seen two kinds of ASSUME/PROVE constructs: one in which the ASSUME clause contains one or more assumptions, and one in which it contains just a NEW declaration. We now examine the precise meaning of this construct.

The statement of the theorem of Figure 2.1 has the form:

(2.17) ASSUME: A, B PROVE P

for formulas A, B, and P. (The numbers attached to A and B are just labels that don't affect the meaning.) This statement asserts the formula $A \wedge B \Rightarrow P$. You can check that the following is a tautology of propositional logic:

$$\models (A \wedge B \Rightarrow P) \;\equiv\; (A \Rightarrow (B \Rightarrow P))$$

[2] I must confess that, many years ago, I used that term in a paper [28].

We can therefore rewrite the formula asserted by (2.17) as $A \Rightarrow (B \Rightarrow P)$. Multiple applications of the tautology show that $A \land B \land C \land D \Rightarrow P$ is equivalent to $A \Rightarrow (B \Rightarrow (C \Rightarrow (D \Rightarrow P)))$, and so on for any number of conjuncts.

In general, an ASSUME clause is a comma-separated list of assumptions, each of which is either a formula or a NEW declaration. Recall that the statement

ASSUME: NEW v PROVE: P

makes the assertion $\forall v : P$. The statement

(2.18) ASSUME: A, B, NEW v, C, D PROVE P

asserts the formula

$$A \Rightarrow (B \Rightarrow (\forall v : C \Rightarrow (D \Rightarrow P)))$$

The proof of the ASSUME/PROVE statement has P as the goal, and the variable v is regarded in the formulas C, D, and P and in any formulas in the proof as a new variable, unrelated to any occurrence of v in A and B and in formulas outside the statement and its proof.

The variable v introduced by NEW v is really a brand new variable, which we think of as having a new unwritten subscript. If A or B has a free variable v_{27}, then the NEW v might be v_{32}. A free variable v in any formula in the proof of P would be v_{32}. This would make it almost impossible to use A or B in the proof if it contained a variable named v. In general, it's a bad idea to introduce a new variable in a context in which a variable with the same name (and a different subscript) has a meaning.

As should be clear from the example, the general form of an ASSUME/PROVE statement is

ASSUME: $A_1, \ldots, A_n$ PROVE: P

where each A_i is either a formula or a NEW declaration. This statement asserts the formula

$$B_1 \ldots B_n \; P\,) \ldots)$$

where B_i is "$A_i \Rightarrow ($" if A_i is a formula and is "$(\forall v :$" if A_i is NEW v. There can be more than one NEW declaration among the A_i. If α is a list of assumptions and P is a formula, we define $AP(\alpha, P)$ to be the formula asserted by the statement ASSUME: α PROVE: P. If α is the empty list, we define $AP(\alpha, P)$ to equal P.

2.1.10.2 Hierarchically Structured Proofs

We now define the precise meaning of a hierarchically structured proof. Let a *statement* be either the statement of the theorem or a statement in the theorem's proof. For each statement and each lowest-level paragraph proof, we define a current goal G and a list β of current assumptions that can be used to prove that goal. For a proof consisting of a list of statements, we consider the current goal and list of assumptions of the first statement of the proof to be the goal and assumptions of the proof.

We recursively define the list of assumptions and the current goal for every statement and paragraph proof in the theorem's proof as follows. We first define them for the theorem. We then define, for every statement, the goal and list of assumptions for the statement's proof and, except for the theorem or a Q.E.D. statement, we also define the goal and list of assumptions for the next statement at the same level. (The next statement at the same level as step number 1.2.3 is step number 1.2.4.) We use the notation that if β and α are lists of assumptions, then β, α is the obvious concatenation of the two lists. If P is a formula, then β, P is the list obtained by appending formula P to the end of β.

We can consider a formula P that is a statement or in a SUFFICES: P statement to be an ASSUME/PROVE with an empty list of assumptions. There are then three kinds of proof statements, where α is a (possibly empty) list of assumptions and P is a formula:

PS1. ASSUME: α PROVE: P

PS2. SUFFICES: ASSUME: α PROVE: P

PS3. Q.E.D.

We now define the goal and list of assumptions for all statements and paragraph proofs, starting with the theorem. The list of assumptions that can be used to prove the theorem is the list of all formulas that have already been proved or are taken as axioms, along with NEW v assumptions for any variables v that have been introduced—usually implicitly. For example, x and y are such variables in the theorem of Figure 2.1. To simplify the definition, we define the current goal where the theorem is stated to equal FALSE. The statement of the theorem must be a formula of type PS1. Here are the inductive definitions for the three kinds of statements, where β is the current list of assumptions and G is the current goal at the statement.

PS1. At the beginning of its proof, the current list of assumptions is β, α, and the current goal is $P \lor G$. (Why that's $P \lor G$ and not P is explained below.) At the next statement, the list of assumptions is $\beta, AP(\alpha, P)$ and the current goal is G.

PS2. At the beginning of the proof, the current list of assumptions is $\beta, AP(\alpha, P)$ and the current goal is G. At the next statement, the list of assumptions is β, α and the current goal is $P \vee G$.

PS3. At the beginning of the proof, the current list of assumptions is β and the current goal is G. (There is no next statement.)

In PS1 and PS2, you probably expected that the stated goals $P \vee G$ should have been P. In fact, P is what is usually proved. However, we can take $P \vee G$ as the goal because the statement that introduces the new goal P lies within a proof of G. If we prove G instead of P, then the rest of the current proof is irrelevant because its purpose is to prove G. Another way to look at it is that we are actually performing a proof by contradiction by adding $\neg G$ as an assumption. If we prove G, then the assumption $\neg G$ allows us to deduce FALSE, which implies the goal P.

The lowest-level steps of a proof have a paragraph proof that mentions all the previous steps needed to prove the step. However, not all previous steps can be used to prove a step. Every step is proved under a list of assumptions, and we know it to be true only if those assumptions hold. Suppose S is the current step and T is a previous step, and suppose β is the list of current assumptions under which S is being proved and α is the list of assumptions under which T has been proved. For T to be used in the proof of S, it must be true under the assumptions of β. Since it was proved under the assumptions of α, we ensure this by requiring that all the assumptions in α must be in β.

Let's suppose for a moment that the proof contains no assumptions in any of the statements of form PS1 or PS2, so all steps in the proof were proved under the same list of assumptions. We could then use any previous statement in the proof of a statement. The proof of step 4.3. could use step 1.1.7.4.5. However, if we allowed that, we would lose the benefit of hierarchical structuring. Hierarchical structuring simplifies understanding because once we've finished proving statements 1, 2, and 3, we can forget about any statements in their proofs when proving statement 4. We just need to understand those three statements. And similarly, when proving statement 4.3, we should not care about the proofs of statements 4.1 and 4.2; we just need to understand statements 1, 2, 3, 4.1, and 4.2. Changes to the proofs of those five statements should make no difference, as long as the statements don't change.

Now suppose there can be assumptions in ASSUMEs. The lists of assumptions under which each of the statements 1, 2, 3, 4.1, and 4.2 were proved are each a prefix of the list of assumptions under which every paragraph proof in the proof of 4.3 is proved. (A list is considered a prefix of the same list.) Respecting the hierarchical structure ensures that only steps that logic allows to be used in a proof can be used. The rule for what statements can then be used in a paragraph proof is:

> **Step Reference Rule** A step in a level-n proof can be used only in the proofs of the steps that follow it in the same level-n proof.

For example, step 6.2.7.3 can be used only in the proofs of steps 6.2.7.4, 6.2.7.5, etc.

This rule allows us to solve the problem of keeping track of long step numbers. Step number 6.2.7.3, 6.2.7.4, etc. can be replaced by the numbers $\langle 4 \rangle 3$, $\langle 4 \rangle 4$, etc., where step $\langle n \rangle i$ is the i^{th} step of a level-n proof. There can be many steps numbered $\langle 4 \rangle 3$ in a proof. However, in any paragraph proof, there is at most one step numbered $\langle 4 \rangle 3$ that the Step Reference Rule allows to be used. That step, if it exists, is the most recent step numbered $\langle 4 \rangle 3$.

2.1.11 Some Useful Notation

Here are two pieces of notation that mathematicians don't seem to need, but that I find essential for writing formulas that describe programs.

2.1.11.1 IF/THEN/ELSE

A programmer who read enough math would notice that mathematicians lack anything corresponding to the **if/then/else** statement of coding languages. Instead, they use either prose or a very awkward typographical convention. We let the expression

$$\text{IF } P \text{ THEN } e \text{ ELSE } f$$

equal e if the predicate P is true and f if P is false. It is defined by:

$$(2.19)\quad \text{IF } P \text{ THEN } e \text{ ELSE } f \;\triangleq\; \text{CHOOSE } v : (P \wedge (v = e)) \vee (\neg P \wedge (v = f))$$

While it's inspired by the **if/then/else** coding language statement, you should not think of an IF/THEN/ELSE expression as instructions for computing something. It is a mathematical expression defined by (2.19). It's more like the expression written in the C language and its descendants as

$P \mathbin{?} e : f$. However, coding languages usually specify the order in which the expressions P, e, and f are evaluated. Although we sometimes say that evaluating an expression *exp* yields a value v, that just means $exp = v$. Formally, there is no concept of evaluation in mathematics.

2.1.11.2 Conjunction and Disjunction Lists

Abstract mathematical descriptions of real systems can be quite long. Definitions are used to decompose them into shorter formulas that are easier to understand. However, those shorter formulas can still be a few dozen lines long. They are understandable because mathematical formulas have a natural hierarchical structure. To take full advantage of that structure, we use a simple bit of notation that mathematicians and many computer scientists find heretical, but that engineers appreciate.

There are two simple ideas: a list of formulas bulleted by $\land$ or $\lor$ represents the conjunction or disjunction, respectively, of those formulas; and indentation is used to replace parentheses. For example, if A, B, ..., J are formulas, then:

$$
\begin{array}{l}
\land \lor A \land B \\
\quad \lor C \\
\land D \Rightarrow E \\
\land \lor \exists x : F \\
\quad \lor \land G \equiv H \\
\qquad \land J
\end{array}
\qquad \text{equals} \qquad
\begin{array}{l}
\quad (\quad (A \land B) \\
\qquad \lor\ C \qquad) \\
\land\ (D \Rightarrow E) \\
\land\ (\quad (\exists x : F) \\
\qquad \lor\ (\quad (G \equiv H) \\
\qquad\qquad \land\ J \qquad) \ )
\end{array}
$$

Note how the implicit parentheses in the bulleted lists delimit the scope of the $\Rightarrow$ and $\exists x$ operators in this formula.

Making indentation significant is a feature of the currently popular Python coding language, but it works even better in this notation because the use of $\land$ and $\lor$ as "bullets" makes the logical structure easier to see.

2.2 Sets

You've probably come across the mathematical concept of a set. A set is a collection. However, we will see that not all collections can be sets. The fundamental operator in terms of which sets are defined is $\in$, which is read *is an element of* or simply *in*. For every collection S that is a set, the formula $exp \in S$ equals TRUE iff the value of the expression *exp* is one of the things the collection S is a collection of. We call the things in a set S the *elements* of S.

We need to be able to have sets of sets—that is, sets whose elements are sets. Therefore, a set has to be a "thing". So, we need to know what kinds of things there are besides sets. The simplest way I know to make the math we need completely rigorous is to base it on what is called ZF set theory or simply ZF, where Z and F stand for the mathematicians Ernst Zermelo and Abraham Fraenkel. One thing that makes ZF simple is that every *thing* is a set. In other words, every mathematical value is a set. We will use ZF to describe our math, so the terms *set* and *value* will mean exactly the same thing. Sometimes I will write *set/value* instead of *set* or *value* to remind you that the two words are synonyms. We add to ZF the operators of predicate logic, so we should say we're using the predicate logic of ZF, but we won't. We'll just call it ZF.

Logicians have shown how to build mathematics, including the set of real numbers, from ZF. There's no need for us to do that; we just assume the real numbers exist and the arithmetic operators on them satisfy the usual properties. This means that 42 and $\sqrt{2}$ are sets, but we don't specify what their elements are. We know that $\sqrt{2} \in 42$ equals either TRUE or FALSE, but we don't know which. We assume nothing about what the elements of the set 42 are. We'll generally use the term *value* for a set/value like 42 for which we don't know what its elements are.

We define the semantics of ZF the same way we defined the semantics of the predicate logic of arithmetic. The meaning $[\![F]\!]$ of a formula F of ZF is a predicate on (the collection of) interpretations, where an interpretation is an assignment of a set to each variable. There is one difference between the predicate logic of arithmetic and ZF: We defined the predicate logic of arithmetic to have two kinds of variables, numeric-valued and Boolean-valued. ZF has just set-valued variables. In ZF, the values TRUE and FALSE are sets.[3] (We don't know what their elements are.)

Including the operators of arithmetic in ZF implies that we have to say what the meaning of the expression $x + y$ is if x or y is a set that isn't a number. It also implies that we can write weird expressions like $(x + y) \land z$, so we have to explain what they mean. We'll deal with this issue in Section 2.2.7.

2.2.1 Simple Operators on Sets

A set S is completely specified by what its elements are—that is, by the values of the formula $e \in S$ for all sets e. This is asserted by the following

[3]As usually defined, ZF does not consider TRUE and FALSE to be sets. We will see that making them sets allows the value of a program variable to be a Boolean.

axiom of ZF:

$$\models (S = T) \;\equiv\; \forall\, e : (e \in S) \equiv (e \in T)$$

Besides equality, set theory has one Boolean-valued operator: the subset operator $\subseteq$. For sets S and T, the formula $S \subseteq T$, read S *is a subset of* T, is true iff every element of S is an element of T. The precise definition is

$$S \subseteq T \;\triangleq\; \forall\, e : (e \in S) \Rightarrow (e \in T)$$

We say that S is a *proper* subset of T iff $S \subseteq T$ and $S \neq T$.

To define a set, we must define what its elements are. The most direct way to do this is by enumerating the elements. If $e_1, \ldots, e_n$ are any values, then they are the (only) elements of the set $\{e_1, \ldots, e_n\}$. This set need not have n elements. For example, the set $\{3, \sqrt{2}, 3, 2{+}1, 42, 3\}$ contains only the three elements $\sqrt{2}$, 3, and 42. It is equal to the set $\{42, 42, 3, \sqrt{2}\}$. (It is as silly to say that a set has two copies of the number 42 as it is to say that a football team has two copies of one of its players.) For $n = 0$, this defines $\{\}$ to be the empty set, which has no elements. For a finite set S (one with only finitely many elements), we let $\#(S)$ be the number of elements of S. For example, $\#(\{1, 2, 4, 2, 1\}) = 3$.

Another way to define a set S is to define what $e \in S$ equals, for a variable e. We do this by writing a formula and asserting that $e \in S$ equals that formula. Here are the names, descriptions, and definitions of the set-valued operators $\cup$, $\cap$, and $\setminus$ (sometimes written "$-$" by computer scientists):

$S \cap T$ (intersection) The set of all elements in both S and T.
Definition: $\models\; e \in S \cap T \;\equiv\; (e \in S) \wedge (e \in T)$

$S \cup T$ (union) The set of all elements in S or T.
Definition: $\models\; e \in S \cup T \;\equiv\; (e \in S) \vee (e \in T)$

$S \setminus T$ (set difference) The set of all elements in S and not in T.
Definition: $\models\; e \in S \setminus T \;\equiv\; (e \in S) \wedge \neg(e \in T)$

Our definition of $S \cap T$ is not just a definition. Besides defining $S \cap T$ to be the collection of values that are in both S and T, it asserts that if S and T are sets, then this collection is also a set. The same is true of the definitions of $\cup$ and $\setminus$ and all the other set-forming operators to be defined. Some of those set-forming operators can be defined in terms of others. For the rest, the assertion that they form a set is an axiom. We don't care which operators are which, but we should be aware that each such definition is making an assertion that some value is a set.

2.2.2 More Sets and Set Operators

We've defined the set $\mathbb{R}$ of all real numbers. Most sets of numbers manipulated by programs are sets of integers. Here are definitions of some sets of integers that we will use.

$\mathbb{N}$ The set of natural numbers, which consists of all the non-negative integers 0, 1, 2, etc.

$\mathbb{I}$ The set of all integers, which includes positive and negative integers and 0.

$m \mathrel{..} n$ If m and n are integers, then this is the set of all integers i such that $m \leq i \leq n$. This means that if $m > n$, then $m \mathrel{..} n$ is the empty set $\{\}$.

We will need two more operators on sets. Even if you've studied sets, you may not be familiar with them:

$\mathcal{P}(S)$ (power set) The set of all subsets of S. For example $\mathcal{P}(\{1,2,3\})$ equals $\{\,\{\,\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\,\}$.
Definition: $\models\ e \in \mathcal{P}(S) \ \equiv\ e \subseteq S$.

$\bigcup S$ The union of all the elements of S. For example, $\bigcup\{T, U, V\} = T \cup U \cup V$.
Definition: $\models\ e \in \bigcup S \ \equiv\ \exists\, s \in S : e \in s$

These operators can be confusing, and when using them it can be hard to keep track of exactly what kind of elements belong to each set. Figuring out why the following two assertions are true may help you understand them.

$$\models \bigcup \mathcal{P}(S) = S \qquad \models S \subseteq \mathcal{P}(\bigcup S)$$

The definitions imply that $\mathcal{P}(\{\})$ equals $\{\{\}\}$, the set whose single element is the empty set, and $\bigcup\{\}$ equals $\{\}$.

2.2.3 Two Set Constructors

ZF has the following two set-forming constructs. I don't think they have names that are used by most mathematicians, so I have chosen these names:

Subsetting $\{v \in S : P\}$ is the subset of the set S consisting of all elements v in S for which formula P is true. For example, $i \mathrel{..} j$ equals $\{n \in \mathbb{I} : i \leq n \leq j\}$.
Definition: $\models\ e \in \{v \in S : P\} \ \equiv\ (e \in S) \land (P \text{ WITH } v \leftarrow e)$

Imaging $\{exp \ : \ v \in S\}$ is the set of all values of the expression *exp* obtained by substituting for v an element of S. For example, $\{2 * n + 1 : n \in \mathbb{N}\}$ is the set of all odd natural numbers.

Definition: $\models \ e \in \{exp \ : \ v \in S\} \ \equiv \ \exists\, v \ : \ (v \in S) \wedge (e = exp)$

These constructs introduce a bound variable, which is named v in the definitions and n in the examples. Like the bound variables of predicate logic, these bound variables have no relation to variables of the same name that might appear elsewhere, and they don't really occur in the expressions.

As with quantifiers, there is an alternative view of these constructs that doesn't involve bound variables. For example, we view the imaging construct $\{exp : v \in S\}$ for an expression *exp* as $Imaging(exp, S)$, where *exp* is a mapping. We view $\{2 * n + 1 : n \in \mathbb{N}\}$ as $Imaging(n \mapsto 2 * n + 1, \mathbb{N})$. I chose the name *Imaging* for this construct because $\{2 * n + 1 : n \in \mathbb{N}\}$ is the set that mathematicians call the image of the set $\mathbb{N}$ under the mapping $n \mapsto 2 * n + 1$.

This view of the imaging construct and the similar view of the subsetting construct shows that the scope of the bound variable v in these constructs is the expression *exp*. It does not include the expression S. For example, the expression $\{v \ : \ v \in \{v\}\}$ is really something like $\{v_{32} : v_{32} \in \{v_{12}\}\}$. If all three of the variables named v were the same variable, then this expression would equal the set of all sets, which we'll see in Section 2.2.6.3 would make the definition unsound because there can't be such a set.

2.2.4 Venn Diagrams

The operators $\cap$, $\cup$, $\subseteq$, and $=$ satisfy the same properties as the operators $\wedge$, $\vee$, $\Rightarrow$, and $\equiv$ of propositional logic. If a propositional logic tautology contains only these four operators, then substituting $\cap$ for $\wedge$, $\cup$ for $\vee$, $\subseteq$ for $\Rightarrow$, and $=$ for $\equiv$ yields a theorem of set theory. Thus $\subseteq$ is transitive; and $\cap$ and $\cup$ are commutative and associative and satisfy these distributivity properties:

$$\begin{aligned}
&\models S \cup (T \cap U) \ = \ (S \cup T) \cap (S \cup U) \\
&\models S \cap (T \cup U) \ = \ (S \cap T) \cup (S \cap U)
\end{aligned}$$

Moreover, if all the sets are subsets of a set $\mathbf{W}$, then a propositional logic tautology containing $\neg$ becomes a theorem of set theory when, in addition to substituting operators of set theory for propositional logic operators, each subexpression $\neg S$ is replaced with $\mathbf{W} \setminus S$.

This correspondence between theorems of propositional logic and set theory is the result of a close connection between predicates and sets. For a set $\mathbf{W}$, there is a natural 1-1 correspondence between predicates on $\mathbf{W}$ and subsets of $\mathbf{W}$. The predicate P and the corresponding set S_P are related as follows:

$$S_P \;=\; \{v \in \mathbf{W} \,:\, P(v)\} \qquad\qquad P(v) \;=\; (v \in S_P)$$

Suppose that instead of letting a value be a ZF set, we let a value be any element of $\mathbf{W}$ and we let a set be any subset of $\mathbf{W}$. The meaning $[\![F]\!]$ of a predicate logic formula F would then be a predicate on $\mathbf{W}$. Define $\overline{\overline{F}}$ to be the subset of $\mathbf{W}$ that corresponds to $[\![F]\!]$, so

$$\overline{\overline{F}} \;\triangleq\; \{v \in \mathbf{W} \;:\, [\![F]\!](v)\}$$

The following calculation shows that $\models \overline{\overline{F \wedge G}} = \overline{\overline{F}} \cap \overline{\overline{G}}$:

$$\begin{array}{rll}
\overline{\overline{F \wedge G}} & = \{v \in \mathbf{W} \,:\, [\![F \wedge G]\!](v)\} & \text{by definition of } \overline{\overline{\ldots}} \\
& = \{v \in \mathbf{W} \,:\, [\![F]\!](v) \wedge [\![G]\!](v)\} & \text{by the meaning of } \wedge \\
& = \{v \in \mathbf{W} \,:\, [\![F]\!](v)\} \cap \{v \in \mathbf{W} : [\![G]\!](v)\} & \text{by definition of } \cap \\
& = \overline{\overline{F}} \cap \overline{\overline{G}} & \text{by definition of } \overline{\overline{\ldots}}
\end{array}$$

Similar calculations show

$$\begin{array}{ll}
\models \overline{\overline{F \vee G}} \;=\; \overline{\overline{F}} \cup \overline{\overline{G}} & \qquad \models \overline{\overline{F \Rightarrow G}} \;=\; (\overline{\overline{F}} \subseteq \overline{\overline{G}}) \\
\models \overline{\overline{F \equiv G}} \;=\; (\overline{\overline{F}} = \overline{\overline{G}}) & \qquad \models \overline{\overline{\neg F}} \;=\; \mathbf{W} \setminus \overline{\overline{F}}
\end{array}$$

If we take $\mathbf{W}$ to be the set of points in a plane, then subsets of $\mathbf{W}$ can be represented by pictures. When used to illustrate predicate logic, such pictures are called Venn diagrams. For example, if predicate symbols F and G are represented by sets of shaded points describing $\overline{\overline{F}}$ and $\overline{\overline{G}}$, then $F \wedge G$ is represented by the points of $\overline{\overline{F}} \cap \overline{\overline{G}}$, where $\overline{\overline{F}}$ and $\overline{\overline{G}}$ overlap. You can find on the Web numerous explanations of the operators of propositional logic using Venn diagrams. Those diagrams provide a good way to become familiar with propositional logic.

2.2.5 Bounded Quantification

In ZF, the formula $\forall\, v : F$ asserts that F is true when any set/value is substituted for v. We usually want to assert only that a formula F is true for all values of v in some set S. This assertion is written $\forall\, v \in S : F$. A similar notation applies to $\exists$ and to the CHOOSE operator. The definitions of these operators are:

(2.20) $$\begin{aligned} \forall\, v \in S : F \;&\triangleq\; \forall\, v : (v \in S) \Rightarrow F \\ \exists\, v \in S : F \;&\triangleq\; \exists\, v : (v \in S) \land F \\ \textsc{choose } v \in S : F \;&\triangleq\; \textsc{choose } v : (v \in S) \land F \end{aligned}$$

We can view $\forall\, v \in S$ and $\exists\, v \in S$ as operators that are the duals of each other. This means that the following assertions are true:

(2.21) $$\begin{aligned} &\models \neg(\forall\, v \in S : F) \;\equiv\; (\exists\, v \in S : \neg F) \\ &\models \neg(\exists\, v \in S : F) \;\equiv\; (\forall\, v \in S : \neg F) \end{aligned}$$

You should convince yourself that they follow from the intuitive meanings of $\forall\, v \in S$ and $\exists\, v \in S$. It's a good exercise to derive them from the definitions (2.20), the duality of $\forall\, v$ and $\exists\, v$ expressed in (2.10), and propositional logic. You should also convince yourself of these properties of quantification over the empty set $\{\}$:

$$\models (\forall\, v \in \{\} : F) \equiv \textsc{true} \qquad \models (\exists\, v \in \{\} : F) \equiv \textsc{false}$$

Some obvious abbreviations are used for nested $\forall$ and nested $\exists$ expressions. For example, $\forall\, v \in S : (\forall\, w \in T : F)$ is written $\forall\, v \in S, w \in T : F$ and $\forall\, v \in S, w \in S : F$ is written $\forall\, v, w \in S : F$. Definitions (2.20) imply:

$$\begin{aligned} &\models (\forall\, v, w \in S : F) \;\equiv\; (\forall\, w, v \in S : F) \\ &\models (\exists\, v, w \in S : F) \;\equiv\; (\exists\, w, v \in S : F) \end{aligned}$$

There are no such abbreviations for CHOOSE. (It's not clear what they would mean.)

The definitions (2.20) imply that the set S lies within the scope of the bound variable v. However, formulas in which v occurs in S are weird. I've never written one, so they don't appear in this book and are not allowed in TLA$^+$.

If S is a finite set containing n elements, then $\forall\, v \in S : F$ equals the conjunction of n formulas, each obtained by substituting an element of S for v in F. Similarly, $\exists\, v \in S : F$ equals the disjunction of those n formulas. If S is an infinite set, we can think of $\forall\, v \in S : F$ and $\exists\, v \in S : F$ as the conjunction and disjunction of the infinitely many formulas obtained by substituting elements of S for v in F. When we say that a formula is a conjunction or disjunction, we sometimes include the case when it is such an infinite conjunction or disjunction. It should be clear from the context when we're doing this.

The rules for reasoning about unbounded quantifiers and unbounded CHOOSE can be applied to the bounded versions of these operators by using the bounded operators' definitions. For example, consider the $\forall$ introduction rule of Section 2.1.9.3. We prove $\forall\, v \in S : F$ by applying the rule to $\forall\, v : (v \in S) \Rightarrow F$, which means proving:

ASSUME: NEW v, $v \in S$ PROVE: F

Since it occurs frequently, we abbreviate NEW v, $v \in S$ as NEW $v \in S$ and write this as:

ASSUME: NEW $v \in S$ PROVE: F

Applying axioms (2.16) to the definition of bounded CHOOSE, using the tautology $\models (R \wedge (P \equiv Q)) \Rightarrow (R \wedge P \equiv R \wedge Q)$, proves:

(a) $\models (\exists\, v \in S : P) \Rightarrow$
$(P \wedge (v \in S)$ WITH $v \leftarrow (\text{CHOOSE } v \in S : P))$

(b) $\models (\forall\, v \in S : P \equiv Q) \Rightarrow$
$((\text{CHOOSE } v \in S : P) = (\text{CHOOSE } v \in S : Q))$

2.2.6 Infinite Sets and Collections

2.2.6.1 Infinite Sets

In an abstract program, the value of a variable might assume any value in an infinite set of possible values. I suspect most people think infinite sets make math more complicated. They're wrong. Math uses infinite sets to make things simpler. One reason coding languages are complicated is they require you to do arithmetic with a finite set of numbers, rather than the infinite set of numbers you used as a child. The simple rules of arithmetic you learned as a child, such as

$$(x + y) + z \;=\; x + (y + z)$$

aren't valid for arithmetic with a finite set of integers.

Infinite sets are needed to make math simple, but they have properties that you may find surprising. An amusing example is called Hilbert's hotel, which is a hotel with an infinite set of rooms numbered 1, 2, 3, etc. Even when Hilbert's hotel is full, there is room for another guest. When a new guest arrives, we just move all guests from their room numbered i to the room numbered $i + 1$, and we give the new guest room number 1.

We say that two finite sets are the same size iff they have the same number of elements. Equivalently, they are the same size iff we can find a 1-1 correspondence between their elements. For example, here is a 1-1 correspondence between the sets $0\,..\,4$ and $\{2, 5, 8, 12, 17\}$ that shows they have the same number of elements.

$$\begin{array}{rccccc} 0\,..\,4: & 0 & 1 & 2 & 3 & 4 \\ & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow \\ \{2,5,8,12,17\}: & 2 & 5 & 8 & 12 & 17 \end{array}$$

We can extend the concept of size of a set to infinite sets by defining any two sets to have the same size iff there exists a 1-1 correspondence between them. One difference between infinite and finite sets is that an infinite set can be the same size as a proper subset of itself. Hilbert's hotel illustrates that adding one element to an infinite set doesn't change its size. Here is a 1-1 correspondence showing that the set of integers is the same size as the set of natural numbers:

$$\begin{array}{rcccccccccc} \mathbb{I}: & 0 & 1 & -1 & 2 & -2 & 3 & -3 & 4 & -4 & \ldots \\ & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \\ \mathbb{N}: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \ldots \end{array}$$

We say that a set S is smaller than a set T iff S is the same size as a subset of T, but T is not the same size as a subset of S. In the 19th century, Georg Cantor upset many mathematicians by showing that the set of integers is smaller than the set of real numbers. He also showed that $\mathcal{P}(S)$ is bigger than S, for every set S. A theorem of ZF called the Schröder-Bernstein Theorem states that if S and T each is the same size as a subset of the other, then S and T are the same size. The definition of size and this theorem generalize to arbitrary collections.

A set is called *countable* iff it is either finite or has the same number of elements as $\mathbb{N}$ (and hence the same number of elements as $\mathbb{I}$). No infinite set is smaller than $\mathbb{N}$, so countably infinite sets are the smallest infinite sets. The following theorem asserts that a countable union of countable sets is a countable set.

Theorem 2.1 If T is a countable set and every element of T is a countable set, then $\bigcup T$ is countable.

PROOF SKETCH: First, assume T and all its elements are infinite sets. Since T is countable, we can enumerate its elements as A, B, C, D, etc. Since all of these sets are countable, we can list their elements as:

$$\begin{array}{lcl} A & = & \{a_0,\, a_1,\, a_2,\, a_3,\, a_4,\, \ldots\} \\ B & = & \{b_0, b_1, b_2, b_3, b_4, \ldots\} \\ C & = & \{c_0,\, c_1,\, c_2,\, c_3,\, c_4,\, \ldots\} \\ D & = & \{d_0,\, d_1,\, d_2,\, d_3,\, d_4,\, \ldots\} \\ \vdots & & \end{array}$$

Here is the required 1-1 correspondence. (The elements of $A \cup B \cup \ldots$ are grouped to make the pattern clearer.)

$$\begin{array}{rcccccccccccc} \mathbb{N}: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \ldots \\ & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \\ A \cup B \cup C \cup D \cup \ldots: & \underbrace{a_0} & \multicolumn{2}{c}{\underbrace{a_1 \quad b_0}} & \multicolumn{3}{c}{\underbrace{a_2 \quad b_1 \quad c_0}} & \multicolumn{4}{c}{\underbrace{a_3 \quad b_2 \quad c_1 \quad d_0}} & \ldots \end{array}$$

If T or any of its elements are finite, we can add elements to them to make them all countably infinite, so we have proved that T is a subset of a countable set. Since $\mathbb{N}$ is the smallest infinite set, any subset of a countable set is countable. End Proof Sketch

2.2.6.2 Mathematical Induction

Any natural numbers n can be written as $(\ldots(0+1)+1)+\ldots)+1$, where there are n ones. This means that we can prove $\forall\, n \in \mathbb{N} : F$ by proving two things:

1. F is true when we substitute 0 for n.
2. If F is true, then it is true when we substitute $n+1$ for n.

Such a proof is called a proof by mathematical induction. We can write these two proof steps as:

1. F with $n \leftarrow 0$
2. Assume: new $n \in \mathbb{N}$, F
   Prove: F with $n \leftarrow n+1$

A little thought shows that we can strengthen the assumption in step 2 to assert not just that F is true for n, but that F is true when any number in $0\,..\,n$ is substituted for n. In other words, we can replace step 2 by:

2. Assume: new $n \in \mathbb{N}$, $\forall\, m \in 0\,..\,n : F$ with $n \leftarrow m$
   Prove: F with $n \leftarrow n+1$

This proof is sometimes called a proof by generalized mathematical induction. We can combine these two steps into one by observing that $0 \mathrel{..} (-1)$ is the empty set, so $\forall\, n \in 0 \mathrel{..} (-1) : F$ equals TRUE. This observation and a little thought shows that generalized mathematical induction can be expressed as proving:

$$\begin{array}{ll} \textsc{Assume:} & \textsc{new } n \in \mathbb{N},\ \ \forall\, m \in 0 \mathrel{..} (n-1) : F \textsc{ with } n \leftarrow m \\ \textsc{Prove:} & F \end{array}$$

Combining the two steps into one saves no work because the proof will be broken into the two cases $n = 0$ and $n > 0$, which are the same as the two steps. However, because $0 \mathrel{..} (n-1)$ is the set of natural numbers less than n, we can rewrite this statement as:

$$(2.22)\quad \begin{array}{ll} \textsc{Assume:} & \textsc{new } n \in \mathbb{N}, \\ & \forall\, m \in \{i \in \mathbb{N} : n > i\} : F \textsc{ with } n \leftarrow m \\ \textsc{Prove:} & F \end{array}$$

Writing induction like this provides a new way of thinking about it. Instead of starting from 0 and going up to bigger numbers, we think of starting from an arbitrary number n and going down to smaller numbers. That is, to prove F is true for n, we assume it's true for numbers m smaller than n. We can then prove F is true for each of those numbers m by assuming its true for numbers p smaller than m. And so on. We can't keep finding smaller and smaller numbers forever. Therefore, we must eventually prove that F is true for some number or numbers without using any assumptions.

The only property of $\mathbb{N}$ necessary for (2.22) to be a sound proof of $\forall\, n \in \mathbb{N} : F$ is that there is no infinite sequence $m_0, m_1, m_2, \ldots$ of elements in $\mathbb{N}$ such that $m_0 > m_1 > m_2 > \ldots$ is true. Statement (2.22) proves $\forall\, n \in S : F$ for any set S with a greater-than relation $>$ satisfying this condition.

For later use, we define the property we need S to satisfy for an arbitrary collection S, not just a set. A *relation on* a collection S is defined to be a Boolean-valued mapping $\succ$ on pairs of values in S, where we write $n \succ m$ instead of $\succ(n, m)$. We define $\succ$ to be *well-founded on* S iff there does not exist a function f with domain $\mathbb{N}$ such that $f(i)$ is in S for all $i \in \mathbb{N}$ and $\forall\, i \in \mathbb{N} : f(i) \succ f(i+1)$. The following theorem can be generalized to an arbitrary collection S, but we state it only for sets.

Theorem 2.2 If $\succ$ is a well-founded relation on the set S, then proving the following statement proves $\forall\, n \in S : F$.

ASSUME: NEW $n \in S$,
$\forall\, m \in \{i \in S \,:\, n \succ i\} \,:\, F$ WITH $n \leftarrow m$
PROVE: F

This kind of proof is called a proof by well-founded induction. An example is proving a property F of subsets of a finite set T by induction on the size of the subset. We prove F is true of the empty set and that it is true of a nonempty subset U of T if it is true of some proper subsets of U. The set S of the theorem is $\mathcal{P}(T)$ and $U \succ V$ is defined to be true iff V is a proper subset of U.

2.2.6.3 Collections

ZF has mind-bogglingly big sets. Not only is $\mathcal{P}(\mathbb{R})$ bigger than $\mathbb{R}$, and $\mathcal{P}(\mathcal{P}(\mathbb{R}))$ bigger than $\mathcal{P}(\mathbb{R})$, and $\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{R})))$ bigger than ... , but the union of all these huge sets is also a set. However, there are collections of sets that are too big to be a set. The biggest such collection is the collection of all sets, which we call $\mathbb{V}$.

Here's how we show that $\mathbb{V}$ can't be a set. If $\mathbb{V}$ were a set, then $\{v \in \mathbb{V} : \neg(v \in v)\}$ would be a set; let's call that set S. Consider the formula $S \in S$. If S were a set, then $S \in \mathbb{V}$ would be true, so the definition of the subsetting construct would imply $S \in S$ is equivalent to $\neg(S \in S)$, which is impossible. Therefore, $\mathbb{V}$ can't be a set. This argument is called Russell's paradox, because it was discovered by the mathematician and philosopher Bertrand Russell.

Since $\mathbb{V}$ can't be a set, any collection $\mathcal{C}$ the same size as $\mathbb{V}$ also can't be a set because if it were, the 1-1 correspondence between the elements of $\mathbb{V}$ and $\mathcal{C}$ would, by the imaging construct, imply that $\mathbb{V}$ is a set. In general, if there is a 1-1 correspondence between the elements of $\mathbb{V}$ and a collection of elements contained in $\mathcal{C}$, then $\mathcal{C}$ can't be a set. (This follows from Russell's paradox and the generalized Schröder-Bernstein Theorem.) For example, the collection of all finite sets isn't a set because $S \leftrightarrow \{S\}$ is a 1-1 correspondence between the values in $\mathbb{V}$ and the collection of sets having one element, all of which are finite sets.

2.2.6.4 Eliminating Collections

There's a way we could use only sets, without mentioning collections that aren't sets. Instead of letting $\mathbb{V}$ be the collection of all sets, we could define it to be a really big set. In particular, we could define it to be big enough to include the real numbers, the Booleans, and all the sets we can express using them and the operators and constructs of ZF. Instead of letting a value be any ZF set, we could let it be any element of $\mathbb{V}$.

The argument of Russell's paradox still applies when $\mathbb{V}$ is defined in this way. It shows that $\mathbb{V}$ can't be an element of itself—in other words, that $\mathbb{V}$ can't be a value. Instead of having collections that are not sets, we would have subsets of $\mathbb{V}$ that are not values.

Changing our definition of $\mathbb{V}$ like this would make no practical difference, because it would make $\mathbb{V}$ big enough to contain every set we would ever want to describe. All it would do is change the terminology, replacing *collection* and *set* with *subset of* $\mathbb{V}$ and *value* (element of $\mathbb{V}$). I think it's less confusing to think about sets versus collections rather than sets that are values versus sets that aren't values. We will therefore keep defining $\mathbb{V}$ to be the collection of all sets.

The possibility of making $\mathbb{V}$ a ZF set is interesting for the following reason. We use math to describe and reason about the collection of all possible executions of a program, and we will see that this collection isn't a set/value. Most math is based on sets, not collections. If we apply math that was developed for reasoning about sets to deduce a result about this collection, we could in theory obtain an invalid result. The math used in this book is, from a mathematical viewpoint, so simple that such a mistake is unlikely. (The one conceivable exception is in Appendix Section A.5.) The ability to make $\mathbb{V}$ a ZF set means that even if we did deduce something that was incorrect because $\mathbb{V}$ is not a set, then a counterexample could not arise in practice.

2.2.7 Meaningless Expressions

We now address the issue, raised above, of what an expression like $(x+y)\wedge z$ means. To do that, we answer a question that might have arisen when you studied arithmetic: What does 1/0 equal?

As a child I was taught that I wasn't supposed to write 1/0. But I later realized that sometimes it's a perfectly natural thing to write. For example, this is a theorem of arithmetic:

$$\models \forall x \in \mathbb{R} : (x \neq 0) \;\Rightarrow\; (x * (1/x) = 1)$$

Therefore, the $\forall$ Elimination law of predicate logic implies that

$$(x \neq 0) \;\Rightarrow\; (x * (1/x) = 1)$$

is true for all $x \in \mathbb{R}$. In particular, this formula is true:

(2.23) $(0 \neq 0) \;\Rightarrow\; (0 * (1/0) = 1)$

This is a true formula containing the expression $1/0$ that children aren't supposed to write.

So, what does $1/0$ equal? Some think that it should be an evil value that can corrupt expressions in which it appears, causing them to equal that evil value. This leads to a three-valued logic, in which the value of a predicate can equal not only TRUE or FALSE but also the evil value. This is unnecessarily complicated.

The simple answer to "What does $1/0$ equal?" is "We don't know." It's a value, but our definition of "/" doesn't tell us what value. It might equal $\sqrt{2}$; it might equal the set $\mathbb{N}$. Formula (2.23) is true because $0 \neq 0$ equals FALSE, and FALSE $\Rightarrow P$ equals TRUE for any predicate P. A formula like $1/0$ is meaningless, meaning that all we know about it is that it's a value.

Meaningless expressions don't concern us because we shouldn't write them. Writing $x \cup 1$ when we meant to write $x + 1$ is an easy error to detect. It will be found with any method that can find subtle errors in abstract programs. There is no reason to complicate the math that we use by trying to make meaningless expressions illegal.

A natural way to define division of real numbers is:

$$r/s \;\triangleq\; \textsc{choose}\ q \in \mathbb{R} : q * s = r$$

Since there is no real number q for which $1 * 0 = 1$, this defines $1/0$ to equal $\textsc{choose}\ q \in \mathbb{R} : \textsc{false}$, which is a meaningless expression. However, $2/0$ equals this same meaningless expression, so this definition implies $1/0 = 2/0$. I don't like that, so I prefer a definition such as:

$$\begin{array}{lll} r/s \;\triangleq\; \textsc{if}\ s \neq 0 & \textsc{then} & \textsc{choose}\ q \in \mathbb{R} : q * s = r \\ & \textsc{else} & \{r\} + \{s\} \end{array}$$

This tells nothing about $r/0$, except that the Substitution Rule implies that it equals $u/0$ if $r = u$.

2.3 Functions

2.3.1 Functions as Mappings

Mathematicians usually define a pair[4] $\langle x, y \rangle$ to be a set (usually the set $\{\{x\}, \{x, y\}\}$) and define a function f to be a set of ordered pairs, where the

[4]Mathematicians generally enclose pairs in parentheses, but parentheses are used for lots of other things, so we make formulas containing pairs easier to read by using angle brackets $\langle$ and $\rangle$ instead.

pair $\langle x, y\rangle$ in f means that $f(x)$ equals y. But they seldom use that definition to define a particular function, instead defining the squaring function sq on real numbers by writing something like:

Let sq be the function on $\mathbb{R}$ such that $sq(x) = x^2$.

I prefer to define a function to be a kind of mapping that is a set/value, without defining what its elements are—in part because it's convenient to define a pair to be a function. The function sq is written as: $x \in \mathbb{R} \mapsto x^2$. In general, the meaning of the function construct $v \in D \mapsto exp$ is defined by this axiom:

$$\models \forall\, e \in D : (v \in D \mapsto exp)(e) \;=\; (exp \text{ WITH } v \leftarrow e)$$

The variable v in this construct is a bound variable, and D is not in its scope. The set D is called the *domain* of the function, and the domain of a function f is written $\text{DOMAIN}(f)$. The value of $f(e)$ is unspecified unless f is a function and e is in its domain. Two functions f and g are equal iff they have the same domain and $f(v) = g(v)$ for all v in their domain. More precisely, we take this to be an axiom:

$$\text{(2.24)}\quad \models ((v \in D \mapsto exp_1) = (v \in E \mapsto exp_2)) \;\equiv\; (D = E) \wedge (\forall\, v \in D : exp_1 = exp_2)$$

We define $D \to S$ to be the set of all function f with domain D such that $f(x) \in S$ for all $x \in D$. A value f is a function with domain D iff f equals $v \in D \mapsto f(v)$. We can therefore define the set $D \to S$ by:

$$\models f \in (D \to S) \;\equiv\; \begin{array}{l} \wedge\ f = (v \in D \mapsto f(v)) \\ \wedge\ \forall\, v \in D : f(v) \in S \end{array}$$

(Like all our definitions of set-forming operators, this asserts that $D \to S$ is a set if D and S are sets.) It follows from (2.24) that there is a unique function whose domain is the empty set. That function can be written $v \in \{\} \mapsto 42$, where we can replace 42 with any value.

An array in modern coding languages is described mathematically as a function, where the expression $f[x]$ in the language means $f(x)$. For a variable f whose value is an array/function, assigning the value 4.2 to $f[14]$ changes the value of f to a new array/function that we can write as

$$x \in \text{DOMAIN}(f) \;\mapsto\; \text{IF } x = 14 \text{ THEN } 4.2 \text{ ELSE } f(x)$$

Mathematicians have little need to write such a function, but it occurs often when math is used to describe programs, so we need a more compact notation for it. We write it like this:

$$f \text{ EXCEPT } 14 \mapsto 4.2$$

This notation has been used above when f is an interpretation Υ. An interpretation is a function whose domain is the set of all variable names.

We have defined what are usually called functions of a single argument. Mathematicians also define functions of multiple arguments. For example, $+$ can be considered to be a function of two arguments, where $x + y$ is an abbreviation of $+(x, y)$. A function with n arguments can be considered to be a function whose domain is a set of n-tuples, so $f(x, y)$ is an abbreviation of $f(\langle x, y \rangle)$.

A function is a special kind of mapping that is a value. A mapping can be defined to assign a specified value to elements of any collection. For example, the mapping $\mathcal{P}$, where $\mathcal{P}(S)$ is the set of subsets of S, is defined on the collection of all sets. However, $\mathcal{P}$ is not a value, so it can't be the value of a variable. More precisely, as explained in Appendix Section A.1, we can't let $\mathcal{P}$ be the value of a variable because we don't know if it's a set. A function is a value, no different than values like $\sqrt{2}$ and $\mathbb{N}$. But the value of $f(v)$ for a function f can be specified only for values v in $\text{DOMAIN}(f)$, which has to be a set. When describing a program mathematically, we often have to decide whether or not to define a particular mapping to be a function. If we want to allow it to be the value of a (mathematical or program) variable, then it must be a function.

2.3.2 Sequences and Tuples

We often number the items in a list, which means giving each item a name like *item number 2*. We will use two different kinds of lists that are numbered in different ways:

Ordinal These are lists whose items are naturally named with the ordinal numbers first, second, third, etc. For example, in a list of people waiting to be served, the second person to be served is naturally named person number 2.

Cardinal These are lists in which it is natural to name an item by its distance from an item numbered 0. For example, it's natural to number the floors of a building by their distance from the ground floor. The ground floor is floor number 0 and floor number 2 is two stories above the ground floor.

I think mathematicians call such lists *sequences*, so we will call them ordinal and cardinal sequences.

In most of today's coding languages, array elements are numbered starting with 0. They make it convenient to describe cardinal sequences and less convenient to describe ordinal sequences. When I started describing abstract concurrent programs with mathematical formulas, I discovered that the formulas were usually simpler if I described finite lists as ordinal sequences. However, the meaning and properties of the formulas are defined in terms of infinite sequences and finite prefixes of those sequences; and the math is simpler if those are cardinal sequences. So, we use both kinds of sequences.

The obvious way to represent a sequence mathematically is as a function whose domain is the set of numbers of the sequence's items. An ordinal sequence of length n (one containing n items) is a function whose domain is $1\,..\,n$. We write ordinal sequences as lists enclosed in angle brackets. Thus, if s is the 4-item ordinal sequence $\langle 2, \{1, \sqrt{2}\}, 2, \langle 7 \rangle\rangle$ then $\textsc{domain}(s) = 1\,..\,4$, $s(1)$ and $s(3)$ equal 2, $s(2)$ equals the set $\{1, \sqrt{2}\}$, and $s(4)$ equals the 1-item sequence $\langle 7 \rangle$, which equals $i \in \{1\} \mapsto 7$.

A pair like $\langle 42, -3\rangle$ is therefore a 2-item sequence. There is no difference between tuples and finite ordinal sequences. An n-tuple is a function with domain $1\,..\,n$. This provides a simple, natural way of referring to the items of a tuple: the i^{th} item of a tuple t is $t(i)$.

The Cartesian product $\times$ of sets is defined so that $S \times T \times U$ is the set of all triples $\langle s, t, u\rangle$ with $s \in S$, $t \in T$, $u \in U$. Deciphering this definition of the general Cartesian product $S_1 \times \cdots \times S_n$ of n sets S_i, as well as the rest of the definitions in this section, is a good way to become familiar with the math that is the topic of this chapter:

$$\begin{aligned}&S_1 \times \cdots \times S_n \;\triangleq\\ &\quad \{\, f \in (1\,..\,n \rightarrow \bigcup\{S_i : i \in 1\,..\,n\}) \;:\; (\forall\, i \in 1\,..\,n \;:\; f(i) \in S_i)\,\}\end{aligned}$$

We define $Seq(S)$ to be the set of finite ordinal sequences whose items are elements of the set S:

$$(2.25)\quad Seq(S) \;\triangleq\; \bigcup\{\,(1\,..\,n \rightarrow S) \;:\; n \in \mathbb{N}\,\}$$

The empty sequence $\langle\,\rangle$ has domain $1\,..\,0$, which is the empty set. It's a simple way to write the (unique) function whose domain is the empty set. (It is the one sequence that is both an ordinal and a cardinal sequence.)

We now define some operators on both ordinal and cardinal sequences. To make it easier to write definitions that apply to both kinds of sequence, for a nonempty sequence s we define $1^{st}(s)$ to equal 1 if s is an ordinal sequence and 0 if it is a cardinal sequence:

$$1^{st}(s) \;\triangleq\; \text{IF } 0 \in \text{DOMAIN}(s) \text{ THEN } 0 \text{ ELSE } 1$$

Recall that $\#(S)$ is the number of elements of a finite set S.

$Len(s)$	The length of the finite sequence s. It equals $\#(\text{DOMAIN}(s))$.
$Head(s)$	The first item of a nonempty sequence s. It equals $s(1^{st}(s))$.
$Tail(s)$	The remainder of the nonempty sequence s after its first item is removed. It equals: $i \in \{\, j-1 \,:\, j \in (\text{DOMAIN}(s) \setminus \{1^{st}(s)\})\,\} \;\mapsto\; s(i+1)$
$s \circ t$	The concatenation of the finite sequence s with the finite or infinite sequence t, both of them the same kind of sequence—either ordinal or cardinal. It equals: $i \in \text{DOMAIN}(s) \cup \{\, j + Len(s) \,:\, j \in \text{DOMAIN}(t)\,\} \;\mapsto\; \text{IF } i \in \text{DOMAIN}(s) \text{ THEN } s(i) \text{ ELSE } t(i - Len(s))$

It would be nice to define $Append(s, v)$ to be the sequence obtained by appending the item v to the end of the finite sequence s. However, there is no way to decide if $Append(\langle\,\rangle, v)$ is an ordinal or cardinal sequence, since the empty sequence $\langle\,\rangle$ is both. We will need to use *Append* only for ordinal sequences, so we define it by:

$$Append(s, v) \;\triangleq\; s \circ \langle v \rangle$$

In this book, a *list* means a piece of syntax consisting of expressions separated by commas. For example:

Let $\mathbf{x}$ be the list $x_1, \ldots, x_n$ of variables.

means that the symbol $\mathbf{x}$ is an abbreviation for $x_1, \ldots, x_n$, where each x_i is a variable. The expression $\langle \mathbf{x} \rangle$ is then an abbreviation for the n-tuple $\langle x_1, \ldots, x_n \rangle$ of variables.

2.4 Definitions

Definitions are omnipresent in mathematics. Despite their importance, mathematicians seem to give little thought to what definitions actually mean. What they mean is a practical concern because if it contained no new definitions, the formula that describes a typical abstract program would be hundreds of lines long. We can understand such a large formula only by using definitions to decompose it into understandable pieces. A precise understanding of the abstract programs we write requires a precise understanding of what those definitions mean.

To understand a defined symbol, we need to understand what the symbol is defined in terms of. If we expand all definitions used in its definition, we obtain a formula containing only operators we assume to be understood by the readers of the formula. All those operators are mappings on collections of values or of tuples (usually pairs) of values. The operators of arithmetic can be viewed as functions whose domains are sets of numbers or pairs of numbers. Operators like $\in$, $\cup$, and DOMAIN are mappings on values/sets or pairs of values/sets. *Operator* just means *mapping.*

The meaning of constructs that introduce bound variables is not hard to define in terms of the view that eliminates bound variables by considering those constructs to be operators on mappings. However, I assume you understand the constructs of ordinary math already defined, and there is no need to define any new ones. So we need only consider definitions of values and operators.

This section first explains the meaning of ordinary definitions, then discusses recursive definitions, and then describes definitions that are local to proofs and to expressions.

2.4.1 Ordinary Definitions

2.4.1.1 Definitions with no Parameters

We define an ordinary (non-recursive) definition to be a syntactic abbreviation. Doing this raises a problem that arises even with the simplest definitions—ones of the form $F \triangleq \ldots$, so F takes no arguments. We illustrate the problem with a simple example.

Suppose that, in a context in which the values of the variables x and y are elements of $\mathbb{R}$, we write the definition $exp \triangleq x + y$. Since x and y are assumed to be real numbers, this defines exp to be a real number. Since exp

and y are real numbers, this formula is valid:

(2.26) $\exists\, x \in \mathbb{R} : x + y > exp$

However, if *exp* is an abbreviation for $x + y$, then formula (2.26) is an abbreviation for

(2.27) $\exists\, x \in \mathbb{R} : x + y > x + y$

which equals FALSE.

The source of this apparent contradiction is what logicians call "variable capture". As explained in Section 2.1.9.1, the variable x in the definition of *exp* and the variable x in (2.26) are actually two different variables. One is the bound variable introduced by $\exists\, x$; the other is the variable x in the definition of *exp*, which was introduced in the context of that definition. So (2.27) is really a lazy way of writing something like:

(2.28) $\exists\, x_{22} \in \mathbb{R} : x_{22} + y > x_{13} + y$

However, unlike formulas we've considered up to now, there's no way to tell by looking at (2.27) that it contains two different variables written as x.

This is not a problem for us now, because we're defining the meaning of the definition; and to define the meaning we can pretend that (2.27) is really (2.28). It's also not a problem when building tools, because those two different variables written x are represented by different *variable* objects. However, it's a problem if we're doing our own reasoning with pencil and paper and want to expand the definition of *exp* in (2.26).

Logicians solve this problem by saying that when you expand the definition of *exp* in (2.26), you have to substitute a different bound variable for x. Instead, I prefer to prevent the problem from arising. This is done by obeying a simple rule:

Never give a new meaning to a symbol that already has a meaning.

For (2.26) to be meaningful, *exp* must already have a meaning. For the definition of *exp* to be meaningful, x must already have a meaning. Hence, the rule forbids the use of x as the bound variable in (2.27). The "already" in the rule assumes formulas are read in logical order, the assigning of a meaning to a symbol preceding the symbol's use. If we wrote (2.26) before we decided what the definition of *exp* should be, the rule would force us to rewrite it if we defined *exp* in terms of a variable named x. This rule is enforced by TLA$^+$ because it helps prevent errors.

However it's done, we will assume that the problem of variable capture has been solved and we will ignore it.

2.4.1.2 Definitions with Parameters

Extending the meaning of a definition to definitions with a parameter is straightforward. The definition $Op(v) = exp$ defines the mapping Op by defining $Op(e)$ for an expression e to be an abbreviation for

(2.29) exp WITH $v \leftarrow e$

The variable v in (2.29) is a bound variable whose scope is the expression exp.

Generalizing to definitions with multiple arguments is also easy. For example, if we define an infix operator $\uplus$ by $w \uplus v \triangleq exp$, then for any expressions $e1$ and $e2$, the expression $e1 \uplus e2$ is an abbreviation for:

$$exp \text{ WITH } w \leftarrow e1,\ v \leftarrow e2$$

Since we regard definitions as purely syntactic abbreviations, they apply just as well to definitions whose arguments need not represent only values but can also represent mappings. For example, this

$$Double(F, x) \triangleq F(F(x))$$

defines $F(\mathcal{P}, \{a, b\})$ to equal $\mathcal{P}(\mathcal{P}(\{a, b\}))$. (This set contains 16 elements, one of which is $\{\{\}, \{b\}\}$.)

2.4.2 Recursive Definitions

A recursive definition is one in which the symbol being defined is used in its definition. Allowing recursive definitions of mappings in ZF without introducing logical inconsistency is tricky. A method of doing this by translating a recursive definition to a non-recursive one was apparently first given in this century [16]. It is explained in Appendix Section A.2. Here we describe recursive definitions informally, showing the translation to a non-recursive definition only for recursive function definitions.

Here is a recursive definition of the operator #, where we have defined # informally to be the mapping such that $\#(S)$ equals the number of elements in S if S is a finite set. (Its value is unspecified if S is an infinite set.)

(2.30) $$\#(S) \triangleq \text{IF } S = \{\} \text{ THEN } 0 \quad \text{ELSE } 1 + \#(S \setminus \{\text{CHOOSE } e : e \in S\})$$

The same kind of reasoning that justifies proof by mathematical induction shows that this defines $\#(S)$ for any finite set S.

In general, for a recursive definition of an operator F on a collection $\mathcal{C}$ of values, the recursion "terminates" and uniquely defines $F(v)$ for all v in $\mathcal{C}$ if there is a well-founded relation $\succ$ on $\mathcal{C}$ for which the following condition is satisfied:

(2.31) For any v in $\mathcal{C}$, the value of $F(v)$ can depend only on the values of $F(w)$ for some values w in $\mathcal{C}$ with $v \succ w$.

Definition (2.30) satisfies condition (2.31), where $\mathcal{C}$ is the collection of all finite sets and $S \succ T$ is defined to equal TRUE iff T is a proper subset of S.

Condition (2.31) allows us to define operators on collections. If we want to define an operator F recursively on a set, then there's a simple way to define F to be a function, if the recursion always terminates. For example, the factorial function *fact* is the function with domain $\mathbb{N}$ such that $fact(n)$ equals 1 if $n = 0$ and $n * (n-1) * \ldots * 1$ if $n > 0$. ($fact(n)$ is usually written $n!$.) Its recursive definition is:

(2.32) $fact \;\triangleq\; n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact(n-1)$

We define this definition to be equivalent to the following non-recursive definition.

$$\begin{aligned} fact \;\triangleq\; & \text{CHOOSE } f : \\ & \qquad f = (n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * f(n-1)) \end{aligned}$$

Remember that CHOOSE $f : P$ equals a value f satisfying P only if $\models \exists f : P$ is true. It is true of the CHOOSE expression in the definition of *fact*, so *fact* does equal the right-hand side of its definition (2.32), because the operator definition

$$f(n) \;\triangleq\; \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * f(n-1)$$

satisfies (2.31), where $\mathcal{C}$ is the set $\mathbb{N}$ and $\succ$ is $>$.

The generalization from this example to arbitrary recursive function definitions should be clear. In practice, it is almost always obvious what a recursive definition of an operator on a collection defines because it's clear that the recursion terminates for all values in the collection. For a function, that collection is its domain, which is a set.

2.4.3 Local Definitions

Thus far, we have considered definitions with no explicit scope. Our definition of *Tail* implicitly remains in effect for the rest of this book. The implicit scope of our definition of *Double* in Section 2.4.1.2 is that section. We now introduce definitions with two explicitly limited scopes.

These local definitions allow the possibility of a symbol's definition occurring in the scope of a definition of the same symbol, or in the scope of a bound variable having the same name. This raises the same problems as multiple variables with the same name, and it can be handled the same way. But in practice, it never occurs for defined symbols.

2.4.3.1 Definitions in Proofs

It's often useful to make definitions local to a proof—more precisely, local to a single proof within a larger hierarchically structured proof. We allow a proof to contain steps that just make one or more definitions. To make the proof easier to read, we begin such a step with the keyword DEFINE. There's no need to number a definition step, since we can refer to the definition by the name of the symbol being defined. As with the assertion of any proof step, the scope of each of the definitions in a DEFINE statement is the rest of the current proof (including the rest of that statement).

A definition needs no proof. However, we may need a proof that a recursive definition defines what we want it to. In that case, the definition step would be followed by a statement asserting this. For example, a recursive function definition would appear in a proof like this:

```
DEFINE $f \triangleq v \in S \mapsto \mathit{FDef}(v, f)$
4.2. $f = (v \in S \mapsto \mathit{FDef}(v, f))$
    4.2.1. $\ldots$
```

See Appendix Section A.2 for how to prove such an assertion. Most of the time, it's obvious that the definition defines what we expect it to, in which case step 4.2 and its proof can be omitted in a hand proof.

2.4.3.2 Definitions in Formulas

We can also make definitions that are local to an individual formula. We do this with a LET/IN construct, the LET clause making definitions that are local to the IN clause. For example, here is the definition of the operator *SetSum*, where *SetSum*(*S*) equals the sum of the elements of any finite set *S* of numbers:

$$\begin{array}{lll} \mathit{SetSum}(S) \triangleq & \text{IF } S = \{\} & \text{THEN } 0 \\ & & \text{ELSE LET } n \triangleq \text{CHOOSE } m : m \in S \\ & & \qquad\ \text{IN } \ n + \mathit{SetSum}(S \setminus \{n\}) \end{array}$$

The symbol n defined in the LET clause is an abbreviation for the CHOOSE expression; its scope is the IN clause. The definition would be equivalent, but harder to read, if the two occurrences of n in the IN clause were replaced by two copies of that CHOOSE expression. A LET clause can contain multiple definitions.

Chapter 3

Describing Abstract Programs with Math

3.1 The Behavior of Physical Systems

Programs are meant to be executed on physical computers. I have been guided by the principle that any statement I make about a program should be understandable as a statement about its execution on one or more computers. I believe this was the principle that guided Turing in defining the Turing machine as an abstraction of a physical computing device.

The description of our science of concurrent programs begins by examining the physics of computing devices. We don't care about the actual details of how transistors and digital circuits work. We are just interested in how scientists describe physical systems. As a simple example, we look at a planet orbiting a star the way an astronomer might.

We consider the one-planet system's behavior starting at some time t_0, after the star and planet have been formed and the planet has settled into its current orbit. Let $\mathbb{R}^{\geq}$ be the set $\{r \in \mathbb{R} : r \geq t_0\}$ of all real numbers r with $r \geq t_0$. The behavior of the one-planet system is described by its state at each instant of time. We assume the star is much more massive than the planet, so we can assume that it doesn't move. We also assume that there are no other objects massive enough to influence the orbit of the planet, so the state of the system is described by the values of six state variables: three describing the three spatial coordinates of the planet's position and three describing the direction and magnitude of its momentum. Let's call those state variables $v_1, \ldots, v_6$; we won't worry about which of the six values each represents. The quantities these variables represent change with time, so the value of each variable v_i is a function, where $v_i(t)$ represents the value at time t. The behavior of the system is described mathematically by the function σ with domain $\mathbb{R}^{\geq}$ such that $\sigma(t)$ is the tuple $\langle v_1(t), \ldots, v_6(t)\rangle$ of numbers, for every $t \in \mathbb{R}^{\geq}$. Physicists call $\sigma(t)$ the *state* of the system at time t.

In this description, the planet is modeled as a point mass. Real planets are more complicated, composed of things like mountains, oceans, and atmospheres. For simplicity, the model ignores those details. This limits the model's usefulness. For example, it's no good for predicting a planet's weather. But models of planets as point masses are sometimes used to plan the trajectories of a real spacecraft. It's also not quite correct to say that the model ignores details like mountains and oceans. The mass of the model's point mass is the total mass of the planet, including its mountains and oceans, and its position is the planet's center of mass. The model abstracts those details, it doesn't ignore them.

The laws that determine the point-mass planet's behavior σ are expressed by six differential equations of this form:

$$(3.1)\quad \frac{dv_i}{dt}(t) \;=\; f_i(t)$$

where $t \in \mathbb{R}^{\geq}$ and each f_i is a function with domain $\mathbb{R}^{\geq}$ such that $f_i(t)$ is a formula containing the expressions $v_1(t), \ldots, v_6(t)$. Don't worry if you haven't studied calculus and don't know what equation (3.1) means. All you need to know is that it asserts the following approximate equality for small non-negative values of dt:

$$(3.2)\quad v_i(t + dt) \;\approx\; v_i(t) + f_i(t) * dt$$

and the approximation gets better as dt gets smaller, reaching equality when $dt = 0$. The differential equations (3.1) have the property that for any time $t > t_0$ and any time $r > t$, the values of the six numbers $v_i(t)$ and the functions f_i completely determine the six values $v_i(r)$ and hence the value of $\sigma(r)$. That is, the equations imply:

History Independence For any time $t \in \mathbb{R}^{\geq}$, the state $\sigma(r)$ of the system at any time $r > t$ depends only on its state $\sigma(t)$ at time t, not on anything that happened before time t.

The generalization from a planetary system to an arbitrary physical system starting at time t_0 is straightforward. The system is described by state variables $v_1, \ldots, v_n$, and its behavior σ is described mathematically as the function with domain $\mathbb{R}^{\geq}$ such that $\sigma(t)$ equals $\langle v_1(t), \ldots, v_n(t)\rangle$. History independence is satisfied by any isolated physical system—that is, by any system that is assumed not to be influenced by anything outside the system.[1]

There is one way our one-planet system differs from most systems. For this system, it is possible to solve the differential equations (3.1) to write the functions v_i as formulas in terms of ordinary mathematical operations. Even for two planets around a star that is not much heavier than them, it is impossible to write such a solution. The functions v_i can be proved to exist and be unique, but the best we can do in general is find very close approximations to those functions for some finite interval of time.

Physics describes systems with math. Remember that in math, there are infinitely many variables. A description of any particular system contains only a finite number of them—the system variables. The description (3.1) of a planet orbiting a star contains only the six system variables describing the planet's state. It doesn't say that there is nothing else in the universe. It just says nothing about any other planets. Instead of thinking of (3.1) and (3.2) as describing a planet orbiting a star, it's more accurate to think of them as describing a universe in which the planet is orbiting the star. They also describe a universe containing both the planet and a spacecraft that orbits the star; they just say nothing about the spacecraft, since the spacecraft is too small to affect the planet's motion. (The spacecraft's motion could be affected by the planet.) Formulas (3.1) and (3.2) just say nothing about the spacecraft.

Physical science is descriptive. The laws of physics describe how a planet moves; they don't instruct the planet. Programs are prescriptive; they tell a computer what to do. This may make it seem strange to use physical science as a guide to a science of programs. But being descriptive or prescriptive is not a property of the math. It's just how we choose to view that math. We can view the equations of planetary motion not only as a description of how a planet moves, but also as commands given to the planet by nature. The math is agnostic. Our science views a program as a description of what behaviors it allows, not as commands for producing those behaviors. This view allows much more freedom in describing programs.

[1]In classical physics, the state at time t_0 uniquely determines the system's subsequent behavior. The situation is less clear in quantum physics where multiple subsequent behaviors seem possible, but the set of those behaviors is completely determined by the state.

3.2 Behaviors of Digital Systems

3.2.1 From Continuous to Discrete Time

Digital systems are physical systems, usually electromagnetic, in which certain stable states represent a collection of one-bit values. For example, at a certain point in a circuit, 0 volts may represent a 0 and 3.3 volts may represent a 1. Classical physics describes the behavior of physical systems as continuous.[2] If the voltage at some point in a circuit changes from 0 volts to 3.3 volts, it must pass through 1 and $\sqrt{2}$ volts.

Computers and other digital systems are designed so that each bit can be thought of as passing instantaneously from one stable state to the next. This means that we can think of there being a sequence of discrete times t_0, t_1, t_2, ... that are the only times at which the value of a bit can change. (We assume there is an event at time t_0 that initializes all the bits of the device.) We pretend that between times t_j and t_{j+1}, the part of the circuit representing each bit is in a stable state. Moreover, whether a bit changes its value at time t_{j+1} depends on the (stable) values of the bits immediately before time t_{j+1}.[3] Thus, the system is history independent.

Although built from one-bit registers, digital systems are designed so that larger components can also be viewed as changing their state instantaneously—for example, a 128-bit register or even all the bits in a chip controlled by a single clock. We can pretend that the entire component changes its value in discrete steps that can occur only at the times t_j. Thus, we can view a digital system as one whose components are represented by state variables that can have more than two values.

When a digital system is executing a program, the state of the program does not correspond directly to the state of the system. The value of a program variable might be represented by different parts of the system at different times. For example, its value may at some times be stored in a memory chip, at some times it may be in a register of a processor chip, and at some times it might be stored on a disk. Later, we'll see what it means mathematically for a digital system to implement a program in this way. For now, consider a concrete program to be just a digital system described

[2]Here, classical physics includes relativity but not quantum mechanics. I believe that quantum mechanics also describes a continuous universe, but a discussion of that would take us too far afield.

[3]The t_j are pretend times, not exact physical times. Two bits that change at the same pretend time may change at different physical times because the clock pulse that generates the change may reach one of them a fraction of a nanosecond before the other. Chip designers must ensure that we can pretend that they change at the same time.

by discretely changing variables whose values are not just bits but may be any data structure provided by the language—for example, 128-bit integers. An abstract program is the same, except the value of a variable may be any value—for example a real number such as $\sqrt{2}$, not just a finite-precision approximation like 1.414213562. Modeling a science of programs on the science of physical systems ensures that it can address real problems, and we are not just creating a science of angels dancing on the head of a pin. (However, the science should be able to describe any discretely behaving angels, wherever they might be dancing.)

We are seldom interested in the actual times t_j at which state variables can change. To simplify things, we consider only the sequence of states through which the system passes, ignoring the times at which it enters and leaves those states. We call the state created at time t_j state number j. Instead of letting a state variable v be a function that assumes the value $v(t)$ at time t, we consider it to be a function that assumes the value $v(j)$ in state number j. In other words, the value of a state variable v is a sequence of values. A behavior σ of a program is also a sequence, where $\sigma(j)$, its state number j, describes the values of the device's variables in that state.

If a program or a digital device runs forever, then the sequence of times t_j is infinite and therefore so is the sequence σ of its states. But if a program terminates, then those sequences can be finite. Other than parallel programs, in which concurrency is added to a traditional program so it can run faster by using multiple processors, most concurrent programs are not supposed to stop. A concrete concurrent program will not really run forever, but we describe it as running forever for the same reason there are an infinite number of integers even though we only use a finite number of them: it makes things simpler.

Still, some concurrent programs are supposed to stop, so we have to describe them. For simplicity, we describe those programs as well with infinite state sequences. Exceptionally observant readers will have noticed that while the times t_j had to be chosen so we can pretend that the state changes only at those times, we did not require that the state *had* to change at each of those times. There can be times t_j at which none of the program variables' values change. In particular, if the program stops, we can add an infinite number of times t_j after it has stopped. This leads to an infinite sequence of states such that, for some k, the values of the program's variables after state number k are the same. We call a pair $\langle \sigma(j), \sigma(j+1) \rangle$ of successive states in a behavior σ a *step* of σ. A step in which the values of the program's variables do not change is called a *stuttering step* of the program.

We call a behavior ending in infinitely many stuttering steps a *halting* behavior of the program. It describes an execution in which the program stops. There are many reasons a program might stop—for example, an error might cause it to abort. If the program stops because it has completed what it was supposed to do, we say that it *terminates*. The term *halting* covers all cases when the program stops.

Mathematically, a behavior of a digital system or an abstract program is an infinite cardinal sequence of states, where each state is an assignment of values/sets to variables. There is a natural tendency to think of state number j of a behavior as occurring at time j on some clock that ticks at a constant rate. Don't think of it like that. A microsecond might elapse between when the system reaches state number j and when it reaches state number $j+1$, and a day or a femtosecond might then elapse before it reaches state number $j+2$. All we know is that the system can't reach state number $j+1$ before it reaches state number j.

By removing any information about the physical time at which things happen, it may appear that we have eliminated the possibility of describing how much actual time it takes for something to happen. That's not the case, and Section 4.4 explains how it's done. However, correctness of few programs depends on exactly how long it takes the program to do something, and I know of no commonly used coding language that allows us to write such programs. To my knowledge, nothing in the definition of the Java coding language assures us that executing the statement `x = x+1` takes less than a century.

3.2.2 An Example: *Sqrs*

Our first example is a very simple abstract program called *Sqrs* that is described in Figure 3.1 with pseudocode. The **variables** statement describes the program variables and their initial values (their values in state 0). In this example, the program variables are x and y, and their initial values are both 1. The program's code consists of a **while** TRUE loop, which means that the body of the loop is repeatedly executed forever. Program *Sqrs* is an abstract program because it runs forever, producing a behavior with an infinite number of states, unlike a concrete program that would halt with an error when x became too big.

In a science, it would be crazy to let "`=`" mean anything other than what it has meant in mathematics for several centuries, so we use "`:=`" to mean assignment. Except for the label a that you can ignore for the moment, it should be obvious what an execution of the loop body does. What's not obvious in most pseudocode and in virtually all real code is how to

```
**variables** $x = 1,\ y = 1$;
**while** TRUE **do**
  $a$: $x \ := x + y + 2$ ;
       $y \ := y + 2$
**end while**
```

Figure 3.1: The simple abstract program *Sqrs*.

represent the execution in a behavior—which means as a sequence of states. In particular, how many different steps in the behavior describe a single execution of the loop body?

We would expect to describe execution of the loop body of *Sqr* with at least one step. But should there be more? For example, should evaluating $x+y$ in the first assignment statement be a separate step? Coding languages seldom answer this question because it makes no difference to the result computed by a traditional program. However, it can make a big difference for concurrent programs.

We will adopt the PlusCal algorithm language's [36] convention of using labels to indicate what the separate steps of a behavior are. The rule is that execution from one label to the next constitutes a single step. This means that a step begins and ends with program execution at a label. For program *Sqrs*, this implies that execution of the entire loop body, starting from label a and finishing when the program reaches a again, is a single step. With this choice of what constitutes a step in the behavior, the values $x(j)$ and $y(j)$ of the variables in each state j of the behavior are determined by two formulas:

(3.3) $(x(0) = 1) \land (y(0) = 1)$

(3.4) $\forall j \in \mathbb{N} : \begin{array}{l} \land\ x(j+1) = x(j) + y(j) + 2 \\ \land\ y(j+1) = y(j) + 2 \end{array}$

We call (3.3) the *initial predicate.* It determines the initial state. Formula (3.4) is called the *step predicate.* It's the discrete analog of the differential equations (3.1) that describe the orbiting planet. Instead of describing how the values of the variables change in the continuous behavior when time increases by the infinitesimal amount dt, the step predicate (3.4) describes how they change when the state number of the discrete behavior increases by one.

You can check that (3.3) and (3.4) define a behavior that begins as follows where, for example, $[x :: 16,\ y :: 7]_3$ indicates that state number 3 assigns the values 16 to x and 7 to y, and the arrows are purely decorative.

$$\begin{bmatrix} x :: 1 \\ y :: 1 \end{bmatrix}_0 \to \begin{bmatrix} x :: 4 \\ y :: 3 \end{bmatrix}_1 \to \begin{bmatrix} x :: 9 \\ y :: 5 \end{bmatrix}_2 \to \begin{bmatrix} x :: 16 \\ y :: \ 7 \end{bmatrix}_3 \to \begin{bmatrix} x :: 25 \\ y :: \ 9 \end{bmatrix}_4 \to \ \cdots$$

These first few states of the behavior suggest that in the complete behavior, x and y equal the following functions:

$$(3.5)\quad \begin{aligned} x &= (j \in \mathbb{N} \mapsto (j+1)^2) \\ y &= (j \in \mathbb{N} \mapsto 2 * j + 1) \end{aligned}$$

To prove that (3.3) and (3.4) imply (3.5), we must prove they imply:

$$(3.6)\quad \forall\, j \in \mathit{Nat} \,:\, (x(j) = (j+1)^2) \ \wedge\ (y(j) = 2 * j + 1)$$

A proof by mathematical induction that (3.3) and (3.4) imply (3.6) is a nice exercise in algebraic calculation.

We can think of (3.5) as the solution of (3.3) and (3.4), just as the formulas describing the position and momentum of the planet at each time t are solutions of the differential equations (3.1). It is mathematically impossible to find solutions to the differential equations describing arbitrary multi-planet systems. It is mathematically possible to write explicit descriptions of variables as functions of the state number like (3.5) for the abstract programs written in practice, but those descriptions are almost always much too complicated to be of any use. Instead, we reason about the initial predicate and the step predicate, though in Section 3.4.1 we'll see how to write them in a more convenient way.

The interesting thing about program *Sqrs* is that it sets the value of x to the sequence of all positive integers that are perfect squares, using only addition. This is obvious from (3.5), but for nontrivial examples we won't have such an explicit description of each state of a behavior. Remember that history independence implies that, at any point in a behavior, what the program does in the future depends only on its current state. What is it about the current state that ensures that if x is a perfect square in that state, then it will equal all greater perfect squares in the future? There is a large body of work on reasoning about traditional programs, initiated by Robert Floyd in 1967 [14], that shows how to answer this question. If you're familiar with that work, the answer may seem obvious. If not, it may seem like it was pulled out of a magician's hat. Obvious or magic, the answer is that the following formula is true for every state number j in the behavior of *Sqrs*:

$$(3.7)\quad \begin{aligned} &\wedge\ (x(j) \in \mathbb{N}) \wedge (y(j) \in \mathbb{N}) \\ &\wedge\ y(j)\,\%\,2 = 1 \\ &\wedge\ x(j) = \left(\frac{y(j)+1}{2}\right)^2 \end{aligned}$$

This formula implies that $x(j)$ is a perfect square, since the first two conjuncts imply that $y(j)$ is an odd natural number. Moreover, since $y(j+1) = y(j) + 2$, the last conjunct implies that $x(j+1)$ is the next larger perfect square after $x(j)$. So, the truth of (3.7) for every state number j explains why the algorithm sets x to all perfect squares in increasing order.

A predicate like (3.7) that is true for every state number j of a behavior is called an *invariant* of the behavior. By mathematical induction, we can prove that a predicate is an invariant by proving these two conditions:

I1. The predicate is true for $j = 0$.

I2. For any $k \in \mathbb{N}$, if the predicate is true for $j = k$ then it's true for $j = k + 1$.

For (3.7), I1 follows from the initial predicate (3.3), and I2 follows from the step predicate (3.4). (Scientists should have no trouble writing the proof; it might be challenging for engineers.)

A predicate that can be proved to be an invariant by proving I1 from an initial predicate and I2 from a step predicate is called an *inductive invariant.* Model checkers can check whether a state predicate is an invariant of small instances of an abstract program. But the only way to prove it is an invariant is to prove that it either is or is implied by an inductive invariant. For any invariant P, there is an inductive invariant that implies P. However, writing an inductive invariant for which we can prove I1 and I2 is a skill that can be acquired only with practice. Tools to find it for you have been developed [15, 38], but I don't know how well they would work on industrial examples.

The first conjunct of the invariant (3.7) asserts the two invariants $x(j) \in \mathbb{N}$ and $y(j) \in \mathbb{N}$. An invariant of the form $v(j) \in S$ for a variable v is called a *type invariant* for v. An inductive invariant almost always must imply a type invariant for each of its variables. For example, without the hypotheses that $x(j)$ and $y(j)$ are numbers, we can deduce nothing about the values of $x(j+1)$ and $y(j+1)$ from the step predicate (3.4).

```
**variables** $x = 1,\ y = 1,\ pc = a$;
**while** TRUE **do**
   $a$: $x\ := x + y + 2$ ;
   $b$: $y\ := y + 2$
**end while**
```

Figure 3.2: The finer-grained abstract program *FGSqrs*.

Most mathematicians would not bother to write the first conjunct of (3.7), simply assuming it to be obvious. However, mathematicians aren't good at getting things exactly right. They can easily omit some uninteresting corner case—for example, the assumption that a set is nonempty. Those "uninteresting corner cases" are the source of many errors in programs. To avoid such errors, we need to state explicitly all necessary requirements, including type invariants.

3.2.3 A Finer-Grained Example: *FGSqrs*

Now consider a modified version *Sqrs* of our abstract program in which the execution of each assignment statement in the body of the **while** loop is represented as a separate step of the behavior. This is specified in the pseudocode by adding a label right before the second assignment statement. The label is b and the program is called *FGSqrs*.

The natural way to describe the state of *FGSqrs* is with the variables x and y and an additional variable to specify which assignment statement is the next one to be executed.[4] Such a variable isn't needed in *Sqrs* because that program has just a single label. The variable we add is traditionally called *pc* (for *p*rogram *c*ounter). We will let its value equal the label from which the execution described by the next step begins. (That execution ends when it reaches the following label.) We assume that a and b are two arbitrary distinct values.

The pseudocode for *FGSqrs* is in Figure 3.2. The **variables** declaration contains *pc* and its initial value, even though we know *pc* is needed because there's more than one label, and program execution is normally assumed to start at the beginning of the code. But, a little redundancy doesn't hurt. A little redundancy doesn't hurt.

[4]In program *FGSqrs*, an additional variable isn't needed because which statement should be executed next can be deduced from the values of the variables x and y, but that's not the case in most programs written in pseudocode.

Here is the mathematical description of the behavior of program *FGSqrs*. As with *Sqrs*, it consists of an initial predicate and a step predicate.

Initial Predicate $(x(0) = 1) \wedge (y(0) = 1) \wedge (pc(0) = a)$

Step Predicate

```
∀ j ∈ ℕ : IF  $pc(j) = a$
              THEN  $\wedge\ x(j+1) = x(j) + y(j) + 2$
                    $\wedge\ y(j+1) = y(j)$
                    $\wedge\ pc(j+1) = b$
              ELSE  $\wedge\ x(j+1) = x(j)$
                    $\wedge\ y(j+1) = y(j) + 2$
                    $\wedge\ pc(j+1) = a$
```

When they see this step predicate, most programmers and many computer scientists think that the conjuncts $y(j+1) = y(j)$ and $x(j+1) = x(j)$ are unnecessary. They think that not saying what the new value of a variable equals should mean that it equals its previous value. But if that were the case, then what we wrote wouldn't be math. We would be giving up the benefits of centuries of mathematical development—the benefits that are the reason science is based on math. An essential aspect of math is that a formula means exactly what it says—nothing more and nothing less. If the step predicate didn't say what $y(j+1)$ equals when $pc(j) = a$ is true, then there would be no more reason for it to equal $y(j)$ than for it to equal $i \in \mathbb{N} \mapsto \sqrt{-42}$.

You may find it discouraging that the mathematical description of *FGSqrs* is more complicated than its pseudocode in Figure (3.2). Please be patient. You will see in Section 3.4.1 how a little notation can simplify it. We can always write an abstract program more compactly in pseudocode than in math, as long as we don't have to explain precisely what the pseudocode means. But science is precise, and a science of abstract programs must explain exactly what they mean. Moreover, tools can't check an imprecise description of a program. Math is the simplest way to explain things precisely.

PlusCal is a precise language for describing abstract programs in what looks like pseudocode. (However, it's infinitely more expressive than ordinary pseudocode because its expressions can be any mathematical expressions—even uncomputable ones.) A PlusCal program is translated to a mathematical description of the program in TLA$^+$. I often find it easier to write an abstract program in PlusCal than directly in TLA$^+$. However, I reason about the TLA$^+$ translation, not the PlusCal code. And for many abstract programs, including most distributed algorithms, it's easier to write the program directly in TLA$^+$ than in PlusCal.

The code whose execution is described as a single step of the behavior is called an *atomic operation*. Because a single step in a behavior describing the execution of *Sqrs* is replaced by two steps in the behavior describing the execution of *FGSqrs*, we say that *FGSqrs* has a *finer grain of atomicity* than *Sqrs*. Having a finer grain of atomicity implies that the step predicate is more complicated.

Having a finer grain of atomicity also implies that the inductive invariant that explains why the abstract program works will be more complicated. However, there is a trick for obtaining the invariant for *FGSqrs* from the invariant (3.7) of *Sqrs*. Define $yy(j)$ to equal $y(j)$ if execution *FGSqrs* is at label a, and to equal the value $y(j)$ will have after executing statement b if execution is at b. The mathematical definition is:

$$yy(j) \;\triangleq\; \text{IF } pc(j) = a \text{ THEN } y(j) \text{ ELSE } y(j) + 2$$

Observe that x and yy are changed at the same time by statement a of *FGSqrs* exactly the same way that the loop body of *Sqrs* changes x and y. Statement b of *FGSqrs* leaves x and yy unchanged. This implies that, because (3.7) is an inductive invariant of *Sqrs*, the formula obtained from (3.7) by substituting yy for y satisfies condition I2 for *FGSqrs*. It's easy to check that this formula also satisfies I1, so it is an inductive invariant of *FGSqrs*. This trick of finding an expression (such as $yy(j)$) that is changed by the fine grained program the way the coarse-grained program changes a variable (such as y) can often be used to obtain an invariant for a finer-grained abstract program from an invariant of a coarser-grained one. It is also at the heart of program refinement, the subject of Chapter 5.

3.3 Nondeterminism

The laws of classical physics, such as the laws of planetary motion, are deterministic. Given the initial values of all the variables, their values at any later time are completed determined. Causes of nondeterminism are either negligible because they have an insignificant effect—for example, meteor showers—or are simply assumed not to happen—for example, cataclysmic collisions with errant asteroids.

Even when executed on supposedly deterministic digital systems, nondeterminism is the norm in programs—especially concurrent ones. Here are some sources of nondeterminism in programs:

User Input The user giving a value to the program is usually described as an action of the program that nondeterministically chooses the value provided by the user. The user can also be described as a separate process that nondeterministically chooses the value to provide.

Random Algorithms Some algorithms can achieve better average performance by making random choices. The science of programs presented here is not meant for describing average properties of possible behaviors, so it can't distinguish this case from one in which random choices are the result of user input.

Generality We may want an abstract program to allow multiple possible implementations. Those possibilities appear as nondeterminism in the abstract program.

Faults Physical devices don't always behave the way they're supposed to. In particular, they can fail in various ways. Programs that tolerate failures describe a failure as an operation that may or may not be executed.

Timing Uncertainty The time taken to perform operations in an individual process can vary from one execution to another for several reasons, including (i) being run on different hardware and (ii) competition for resources with other processes in the same program or in concurrently executed programs. This results in multiple behaviors in which operations in different processes are executed in different orders. Those different orders can lead to very different behaviors.

Timing uncertainty is the most important source of errors due to nondeterminism that affects all concurrent programs (not just fault-tolerant ones). Let's examine a simple example of it.

The example is a trivial abstract multiprocess program called *Increment*. It has a variable x that initially equals 0, and each process just increments x by 1 and terminates. A process does this in two steps: the first step reads the current value of x, and the second step sets x to one plus the value it read. You should convince yourself that with N processes, an execution can terminate with x having any value from 1 through N. The final value of x will be N if each process executes its two steps with no intervening step by any other process. The final value will be 1 if all processes read x before any process sets the value of x.

This abstract program is described with pseudocode in Figure 3.3, where *Procs* is the set of processes. (*Procs* is really a set of process identifiers,

```
**variables** $x = 0$ ;
**process** $p \in Procs$
  **variables** $t = 0$, $pc = a$ ;
    $a$: $t := x$ ;
    $b$: $x := t + 1$
**end process**
```

Figure 3.3: The *Increment* abstract program for a set *Procs* of processes.

but for convenience we call its elements processes.) The only assumption we make about this set is that it is finite and nonempty. The **process** statement declares that there is a process for every element of *Procs*, and it gives the code for an arbitrary process p in *Proc*. The variables t and pc are local to process p, each process having its own copy of these two variables. Variable x is global, accessed by all the processes. Process p saves the result of reading x in its variable t. The initial value of t doesn't matter, but letting all variables have reasonable initial values makes a type invariant simpler, so we let t initially equal 0.

The mathematical description of the abstract program *Increment* is in Figure (3.4). The process-local variables t and pc are represented by mathematical variables whose values in each state are functions with domain *Procs*, where $t(p)$ and $pc(p)$ are the values of those variables for process p. The initial predicate, describing the values of the variables in state number 0, is simple. The possible steps in a behavior are described by a predicate that, for each j, gives the values of $x(j+1)$, $t(j+1)$, and $pc(j+1)$ for any assignment of values to $x(j)$, $t(j)$, and $pc(j)$. It asserts that there are two possibilities, described by formulas $PgmStep(j)$ and $Stutter(j)$, that are explained below.

$PgmStep(j)$ describes the possible result of some process executing one step starting in state j. The predicate equals true iff there exists a process p for which $aStep(p, j)$ or $bStep(p, j)$ is true, where:

$aStep(p, j)$ describes a step in which process p executes its statement labeled a in state number j. Its last three conjuncts describe the values of the three variables x, t, and p in state $j+1$. Many people are tempted to write $t(j+1)(p) = x(j)$ and $pc(j+1)(p) = b$ instead of the third and fourth conjuncts. But that would permit $t(j+1)(q)$ and $pc(j+1)(q)$ to equal any values for $q \neq p$. Instead we must use the EXCEPT operator defined in Section 2.3.1. The

Initial Predicate

$$\begin{array}{ll} \land\ x(0) & = 0 \\ \land\ t(0) & = (p \in \mathit{Procs} \mapsto 0) \\ \land\ pc(0) & = (p \in \mathit{Procs} \mapsto a) \end{array}$$

Step Predicate

$$\forall j \in \mathbb{N} : \mathit{PgmStep}(j) \lor \mathit{Stutter}(j)$$

where

$$\begin{array}{lll} \mathit{PgmStep}(j) & \triangleq & \exists p \in \mathit{Procs} : \mathit{aStep}(p,j) \lor \mathit{bStep}(p,j) \\[6pt] \mathit{aStep}(p,j) & \triangleq & \begin{array}{lll} \land\ pc(j)(p) & = & a \\ \land\ x(j+1) & = & x(j) \\ \land\ t(j+1) & = & (t(j) \text{ EXCEPT } p \mapsto x(j)) \\ \land\ pc(j+1) & = & (pc(j) \text{ EXCEPT } p \mapsto b) \end{array} \\[6pt] \mathit{bStep}(p,j) & \triangleq & \begin{array}{lll} \land\ pc(j)(p) & = & b \\ \land\ x(j+1) & = & t(j)(p)+1 \\ \land\ t(j+1) & = & t(j) \\ \land\ pc(j+1) & = & (pc(j) \text{ EXCEPT } p \mapsto \mathit{done}) \end{array} \\[6pt] \mathit{Stutter}(j) & \triangleq & \begin{array}{l} \land\ \forall p \in \mathit{Procs} : pc(j)(p) = \mathit{done} \\ \land\ \langle x(j+1), t(j+1), pc(j+1) \rangle = \langle x(j), t(j), pc(j) \rangle \end{array} \end{array}$$

Figure 3.4: The *Increment* abstract program in math.

first conjunct is a predicate that is true or false of state j. It is an *enabling condition*, allowing the step described by the following three conjuncts to occur iff that condition is true.

$\mathit{bStep}(p,j)$ describes a step in which process p executes its statement labeled b in state number j. It is similar to $\mathit{aStep}(p,j)$. Its enabling condition is $pc(j)(p) = b$. The step sets $pc(j+1)(p)$ to *done*, which is a value indicating that the process has reached the end of its code and terminated.

$\mathit{Stutter}(j)$ describes a stuttering step starting in state j. It is enabled iff $pc(j)(p)$ equals *done* for all $p \in \mathit{Procs}$, so all processes have terminated. At that point, $\mathit{PgmStep}(j)$ is not enabled, so only an infinite sequence of stuttering steps can occur, as required for a terminated abstract program. Note that the second conjunct in the definition of $\mathit{Stutter}(j)$ uses the fact that two tuples are equal iff their corresponding elements are equal to write the following formula more compactly:

$$(x(j+1) = x(j)) \land (t(j+1) = t(j)) \land (pc(j+1) = pc(j))$$

A property we might like to prove about abstract program *Increment* is that, when it has terminated, the value of x lies between 1 and the number of processes. Let's define N to equal $\#(Procs)$, the number of processes. Since a process has terminated iff its local pc variable equals *done*, the property we want to prove is that this formula is an invariant of *Increment*—that is, true for every $j \in \mathbb{N}$:

(3.8) $(\forall p \in Procs : pc(j)(p) = done) \Rightarrow (x(j) \in 1 \,..\, N)$

This is not an inductive invariant because condition I2 is not satisfied. For example, suppose the following is true:

- $pc(j)(p) = b$ and $pc(j)(q) = done$ for all $q \neq p$
- $t(j)(p) = N$

Then (3.8) is true in state number j, but false in state number $j+1$.

To show that (3.8) is an invariant of *Increment*, we must find an inductive invariant that implies it. Stopping now and trying to find that inductive invariant by yourself is a good exercise. But it's not easy if you don't have practice finding inductive invariants and don't have a tool to check if what you think is an inductive invariant actually is one. So, I will write one for you.

An inductive invariant almost always requires a type invariant for each variable. We start by defining *TypeOK* to assert a type invariant for each of the three variables:

$$\begin{aligned} TypeOK(j) \;\triangleq\; & \land\; x(j) \in 0 \,..\, N \\ & \land\; t(j) \in (ProcSet \rightarrow 0 \,..\, N) \\ & \land\; pc(j) \in (ProcSet \rightarrow \{a, b, done\}) \end{aligned}$$

TypeOK is an invariant, but not an inductive invariant. For example, if $x(j) = 1$, $t(j)(p) = N$, and $pc(j)(p) = b$, then $TypeOK(j)$ is true but a step satisfying $bStep(p, j)$ makes $TypeOK(j+1)$ false. We can make *TypeOK* an inductive invariant by weakening it, replacing the two occurrences of $0 \,..\, N$ with $\mathbb{N}$. However, I prefer a stronger, more informative type invariant.

To write the rest of the inductive invariant, we define $NumberDone(j)$ to be the number of processes that have terminated in state j. The precise definition is:

$$NumberDone(j) \;\triangleq\; \#(\{p \in Procs : pc(j)(p) = done\})$$

The complete inductive invariant, which we call *Inv*, is defined by:

$$(3.9)\quad Inv(j) \;\triangleq\; \begin{array}{l} \land\ TypeOK(j) \\ \land\ \forall\, p \in Procs : \\ \qquad (pc(j)(p) = b) \;\Rightarrow\; (t(j)(p) \leq NumberDone(j)) \\ \land\ x(j) \leq NumberDone(j) \end{array}$$

To prove that *Inv* is an inductive invariant of program *Increment*, we must prove I1 and I2. I1 asserts that the initial predicate implies $Inv(0)$, and I2 asserts that the step predicate implies $Inv(j) \Rightarrow Inv(j+1)$. We will not consider how these conditions are proved until we have a more convenient way of writing them.

3.4 Temporal Logic

3.4.1 The Logic of Actions

3.4.1.1 Eliminating State Numbers

There's an easy way to simplify initial predicates, step predicates, and invariants: remove the explicit state numbers. It's obvious that an initial predicate is about state number 0, so we can eliminate every "(0)" in it. An invariant is true for all states, so we don't have to say which states it's about. For a step predicate, we just have to distinguish between $v(j)$ and $v(j+1)$ for a variable v. A notation for doing this that dates back at least to the early 1980s is to replace $v(j)$ by v and $v(j+1)$ by v'. The initial and step predicates of program *Increment* have been rewritten this way in Figure 3.5, where they've been given the names *Init* and *Next*. The inductive invariant (3.9). is also rewritten without the "(j)" and named *Inv*. (The "(t)" has been implicitly removed from the definition of *NumberDone*.) Make sure that you understand Figure 3.5 by comparing it with Figure 3.4.

Mathematically, the big leap from Figure 3.4 to Figure 3.5 is removing the explicit mention of state numbers—for example, writing x instead of $x(j)$. In Figure 3.4, *Procs* and x are both ordinary mathematical variables. The value of *Procs* is a set of processes and the value of x is a function whose domain is *Nat*. In Figure 3.5, the value of *Proc* is a set of processes—the same set throughout a behavior of the process. However, the value of x depends on the state of the behavior.

The price of removing explicit state numbers from our formulas is leaving the domain of ordinary math, with a single kind of variable, and entering a new kind of math in which there are two kinds of variables: mathematical

Initial Predicate

$$\begin{array}{lll} \mathit{Init} & \triangleq & \land\ x\ = 0 \\ & & \land\ t\ = (p \in \mathit{Procs} \mapsto 0) \\ & & \land\ pc = (p \in \mathit{Procs} \mapsto a) \end{array}$$

Step Predicate

$$\mathit{Next} \triangleq \mathit{PgmStep} \lor \mathit{Stutter}$$

where

$$\begin{array}{lll} \mathit{PgmStep} & \triangleq & \exists\, p \in \mathit{Procs} : \mathit{aStep}(p) \lor \mathit{bStep}(p) \\ \mathit{aStep}(p) & \triangleq & \land\ pc(p) = a \\ & & \land\ x' \ = x \\ & & \land\ t' \ = (t \text{ EXCEPT } p \mapsto x) \\ & & \land\ pc' = (pc \text{ EXCEPT } p \mapsto b) \\ \mathit{bStep}(p) & \triangleq & \land\ pc(p) = b \\ & & \land\ x' \ = t(p) + 1 \\ & & \land\ t' \ = t \\ & & \land\ pc' = (pc \text{ EXCEPT } p \mapsto \mathit{done}) \\ \mathit{Stutter} & \triangleq & \land\ \forall\, p \in \mathit{Procs} : pc(p) = \mathit{done} \\ & & \land\ \langle x', t', pc' \rangle = \langle x, t, pc \rangle \end{array}$$

Inductive Invariant

$$\begin{array}{lll} \mathit{Inv} & \triangleq & \land\ \mathit{TypeOK} \\ & & \land\ \forall\, p \in \mathit{Procs} : (pc(p) = b) \Rightarrow (t(p) \leq \mathit{NumberDone}) \\ & & \land\ x \leq \mathit{NumberDone} \end{array}$$

Figure 3.5: Abstract program *Increment* and its invariant *Inv* in simpler math.

variables like *Procs*, whose values are the same in every state of a behavior, and program variables like x that are implicit functions of the state. Program variables like x look weird to mathematicians. In math, the value of a variable x is fixed. We've seen in Chapter 2 that when a mathematician does something else and introduces a variable x, it's really a completely different variable that happens also to be written "x". Of course, you're familiar with program variables because they're the variables of coding languages, whose values change in the course of a computation.

Since this book is about a science of programs, we will henceforth use the name *variable* for program variables. Mathematical variables like *Procs* will be called *constants*. When describing a program mathematically, variables correspond to what we normally think of as program variables. Constants are parameters of the program, such as a fixed set of processes. Early coding languages had constants as well as variables. In modern coding languages, constants are buried in the code, where they are called static final variables of an object.

In this book, the variables in pseudocode are explicitly declared, and undeclared identifiers like *Procs* are constants. For formulas, the text indicates which identifiers are variables and which are constants.

In addition to having both variables and constants, the formulas in Figure 3.5 have primed variables, like x'. An expression that may contain primed and unprimed variables, constants, and the operators and values of ordinary math (which means everything described in Chapter 2) is called a *step expression*. A Boolean-valued step expression is called an *action*. The math whose formulas are actions is called the Logic of Actions, or LA for short.

3.4.1.2 The Semantics of the Logic of Actions

As we did in defining the semantics of elementary algebra in Section 2.1.5, we define the meaning $[\![exp]\!]$ of an expression of LA to be a mapping on interpretations. An interpretation assigns values to variables. Since LA has both constants and variables, there are two parts to an interpretation: an assignment of values to constants and an assignment of values to variables.

Since constants are ordinary mathematical values, and we have already discussed the semantics of ordinary math, we will ignore the part of an interpretation for LA that assigns values to them. When discussing a formula of LA, we assume that there is some fixed interpretation Υ that assigns values to the constants. Constants are usually assumed to satisfy some conditions. For example, the constants M and N of Euclid's algorithm in Section 1.5

are assumed to be positive integers, and the constant *Procs* of program *Increment* in Section 3.3 is assumed to be a finite set. We assume that the fixed interpretation Υ satisfies those assumptions. We define $\models F$ for a formula F of LA to mean that $[\![F]\!]$ is true for all interpretations in which the assignment of values to the constants satisfies the assumptions.

In LA, there are effectively two kinds of variables: unprimed and primed. An interpretation of LA assigns values to each of those kinds of variables, where the values assigned to v and v' are independent of one another.

We have defined a state to be an assignment of values to program variables. So, since we're neglecting constants, an interpretation for an LA formula is a pair of states—the first assigning values to the unprimed variables and the second assigning values to the primed variables. We have used the term *step* to mean a pair of successive states in a behavior. We now let it mean any pair of states. We will write the step consisting of the states s and t as $s \rightarrow t$ rather than $\langle s, t \rangle$ because that makes it clear that s and t are states.

To define the semantics of LA, we therefore have to define $[\![exp]\!](s \rightarrow t)$ for any states s and t. We have not yet defined any operators for LA, so the only operators that can appear in an LA expression are ordinary mathematical operators like $+$ and *Tail*. They have the usual semantics in LA. For example

$$[\![exp1 + exp2]\!](s \rightarrow t) \;\triangleq\; [\![exp1]\!](s \rightarrow t) + [\![exp2]\!](s \rightarrow t)$$

For an unprimed variable v, we define $[\![v]\!](s \rightarrow t)$ to equal $s(v)$, the value assigned to variable v by state s. For a primed variable v', we define $[\![v']\!](s \rightarrow t)$ to equal $t(v)$.

We call an LA expression a *step expression* and an LA formula an *action*. For an action A and step $s \rightarrow t$, we say that $s \rightarrow t$ *satisfies* A or is an A *step* iff $[\![A]\!](s \rightarrow t)$ equals TRUE.

A *state expression* is an LA expression that contains no primed variables, and a *state formula* is a Boolean-valued state expression. For a state expression *exp*, the value of $[\![exp]\!](s \rightarrow t)$ depends only on s, so we can write it as $[\![exp]\!](s)$.

Because the meaning of an LA expression assigns different values to v and v', we can treat v and v' as two unrelated variables. This means that we can reason about LA formulas as if constants, unprimed variables, and primed variables were all different mathematical variables. Thus, for LA as defined so far, we can regard LA as ordinary math with some mathematical variables having names like v' that ending with $'$.

3.4.1.3 The Prime Operator ($'$)

In Figure 3.5, only variables are primed. In the Logic of Actions, we can prime not just a variable but any state expression—that is, any expression containing no primes. For a state expression *exp*, the value of the step expression exp' on a step $s \rightarrow t$ is the value of *exp* on s. More precisely, the meaning of exp' is defined by $[\![exp']\!](s \rightarrow t) = [\![exp]\!](t)$. This means that exp' is equivalent to the step expression obtained by priming all the variables in *exp*. The priming operator ($'$) can be applied only to state expressions. In LA, priming an expression that contains a prime is a syntax error. That means that it is illegal to prime an expression containing a defined symbol whose definition contains a prime. For example, if e is defined to equal $x' + 1$, then e' is syntactically illegal.

A constant has the same value in both states of a step. Therefore, $\models c' = c$ is true for any constant c. More generally, a *constant expression* is an expression with no (primed or unprimed) variable; and $\models exp' = exp$ is true for any constant expression *exp*. The bound identifiers of predicate logic are like ordinary mathematical variables, which means they are treated like constants in the Logic of Actions. For example, $(\exists\, i \in Nat : y = x + i)'$ equals $\exists\, i \in Nat : y' = x' + i$. We therefore call bound identifiers *bound constants*. Appendix Section A.3 gives an example of how you can get into trouble by forgetting that bound identifiers are constants.

The semantics of LA imply that the prime operator distributes over the operators and constructs of ordinary math—for example, that $(S \cup T)'$ equals $S' \cup T'$. By expanding all definitions and distributing primes in this way, we obtain a formula in which the prime operator is applied only to variables. We don't have to expand all definitions to obtain such a formula. We need only expand definitions that contain a prime or that appear within a primed expression and contain a variable. Once we have reached an expression in which only variables are primed, we can reason about the resulting expression as if constants, variables, and primed variables were all ordinary mathematical variables. We therefore need no additional rules for reasoning about LA formulas.

Section 3.2.2 defined an inductive invariant *Inv* of a program to be a state predicate satisfying conditions I1 and I2, which we can restate as:

I1. *Inv* is implied by the program's initial state.

I2. If *Inv* is true in a state, then the program's next-state predicate implies that it is true in the next state.

For program *Increment*, whose initial predicate is *Init* and whose next-state action is *Next*, these two conditions can be expressed in LA as:

$$\begin{aligned} (3.10) \quad & \models \mathit{Init} \Rightarrow \mathit{Inv} \\ & \models \mathit{Inv} \land \mathit{Next} \Rightarrow \mathit{Inv}' \end{aligned}$$

The proof of these conditions for program *Increment* is discussed in Appendix Section B.1.

Thus far, the correctness properties of programs that have concerned us have been invariance properties. All the reasoning we have done to verify that a program satisfies an invariance property is naturally expressed in LA. The safety property usually proved of a traditional program is that it cannot produce a wrong answer—which is expressed as the invariance of the property asserting that the program has not terminated with a wrong answer. The most popular way of proving such a property is Hoare logic [21]. Appendix Section A.4 explains Hoare logic and its relation to the Logic of Actions.

3.4.1.4 Action Composition

The Logic of Actions contains another operator that is almost never used in describing abstract programs and will not play a major role for us until Section 7.1. However, it does make brief appearances in Sections 4.3.2 and 5.4.4.3, so it is explained here.

For actions A and B, the action $A \cdot B$ is defined to be true of a step $s \rightarrow t$ iff there is a state u such that $s \rightarrow u$ is an A step and $u \rightarrow t$ is a B step. If actions A and B describe two pseudocode statements S_a and S_b, then $A \cdot B$ describes the statement $S_a; S_b$ executed by executing S_a followed by S_b. For example, the statements labeled a and b in process p of program *Increment* shown in Figure 3.3 are described by actions $\mathit{aStep}(p)$ and $\mathit{bStep}(p)$ of Figure 3.5, and:[5]

$$\mathit{aStep}(p) \cdot \mathit{bStep}(p) \;\equiv\; \begin{array}{l} \land\; pc(p) \;=\; a \\ \land\; x' \;\;=\;\; t(p) + 1 \\ \land\; t' \;\;=\;\; (t \text{ EXCEPT } p \mapsto x) \\ \land\; pc' \;=\; (pc \text{ EXCEPT } p \mapsto \mathit{done}) \end{array}$$

Replacing the actions $\mathit{aStep}(p)$ and $\mathit{bStep}(p)$ with $\mathit{aStep}(p) \cdot \mathit{bStep}(p)$ in the definition of the next-state action *Next* of program *Increment* produces a

[5]If you believe that the second and third conjuncts in this formula are in the wrong order, then you're thinking in terms of coding languages, not math. Remember that $F \land G$ is equivalent to $G \land F$.

program with a coarser grain of atomicity. Choosing the grain of atomicity of an abstract program involves a tradeoff between making the program detailed enough to be useful and simple enough to be usable. Section 7.1 addresses this tradeoff using action composition.

The operator "$\cdot$" is associative, meaning $(A \cdot B) \cdot C = A \cdot (B \cdot C)$ for any actions A, B, and C. We can therefore omit parentheses and simply write $A \cdot B \cdot C$.

For any action A, we define the action A^+ to be satisfied by a step $s \rightarrow t$ iff state t can be reached from state s by a sequence of one or more A steps. In other words:

$$A^+ \;\triangleq\; A \,\lor\, (A \cdot A) \,\lor\, (A \cdot A \cdot A) \,\lor\, (A \cdot A \cdot A \cdot A) \,\lor\, \ldots$$

3.4.2 The Temporal Logic RTLA

In 1977, Amir Pnueli [46] had the idea of using an obscure branch of mathematics called temporal logic to express time-dependent properties without explicitly mentioning times or state numbers. He used a temporal logic containing the single temporal operator $\Box$ and operators defined in terms of $\Box$. You can read $\Box$ as *always*, but when you become more familiar with it you'll probably just call it *box*. Intuitively, the formula $\Box F$ asserts that the formula F is true at all times. For example, if P is a state predicate, then $\Box P$ asserts that P is true in all states of a behavior.

From now on, we will be discussing and using temporal logic. We will continue to ignore assignments of values to constants, assuming some fixed interpretation satisfying the assumptions made about those constants.

In the kind of temporal logic Pnueli used, called *linear-time* temporal logic, the meaning of a formula is a predicate on behaviors, where a *behavior* is an infinite cardinal sequence of states. In other words, a behavior σ is a function from $\mathbb{N}$ to states. We think of $\sigma(n)$ as the state at time n, so the first state of σ is $\sigma(0)$. But remember, the only resemblance of the state number n to a time is that state $\sigma(n)$ does not occur later than state $\sigma(n+1)$. (In Section 4.4, we'll see that they could both occur at the same time.) We'll sometimes write σ as $\sigma(0) \rightarrow \sigma(1) \rightarrow \cdots$.

RTLA is the temporal logic containing the same temporal operators as Pnueli's original logic, all defined in terms of $\Box$, but having the formulas of LA as the basic formulas. The formulas of RTLA can all be written as LA formulas and formulas obtained from them using the operator $\Box$ and the usual operators of propositional and predicate logic. The only expressions of RTLA are formulas. Prime ($'$) can appear only in the basic LA formulas. It's illegal to prime a formula containing a $\Box$.

The TLA in RTLA stands for *Temporal Logic of Actions.* The R stands for *Raw*, in the sense of *unrefined.* We'll see later that RTLA allows us to write formulas that we shouldn't write. TLA is the logic obtained by restricting RTLA to make it impossible to write those formulas. But that's a complication we don't need to worry about now, so we'll start with the simpler "raw" logic.

In temporal logic formulas, the operator $\Box$ binds more tightly than the operators of propositional logic. For example, $\Box F \vee G$ is parsed as $(\Box F) \vee G$.

3.4.2.1 Simple RTLA

In RTLA, the operator $\Box$ can be applied to any RTLA formula, so we can write formulas like $\Box(A \Rightarrow \Box B)$ where A and B are actions. We will begin by considering simple RTLA, in which the operator $\Box$ is applied only to actions, not to formulas containing $\Box$.

A formula of a temporal logic is called a temporal formula. For any assignment of values to constants, the meaning $[\![F]\!]$ of a temporal formula is a behavior predicate—that is, a mapping that assigns Boolean values to behaviors. An action A is a formula of RTLA, where it is viewed as a behavior predicate. As a formula of LA, we've viewed A as a step predicate. As a formula of RTLA, we view it as a behavior predicate that is true on a behavior iff, viewed as a step predicate, it is true of the behavior's first step.

To state that precisely, let $[\![A]\!]_{LA}$ be the meaning of A as an LA formula. We define its meaning $[\![A]\!]_{RTLA}$ as an RTLA formula as follows. For any behavior σ, which equals $\sigma(0) \rightarrow \sigma(1) \rightarrow \sigma(2) \rightarrow \cdots$, we define

(3.11) $[\![A]\!]_{RTLA}(\sigma) \;\triangleq\; [\![A]\!]_{LA}(\sigma(0) \rightarrow \sigma(1))$

From now on, $[\![F]\!]$ means $[\![F]\!]_{RTLA}$ for all RTLA formulas, including actions. We will explicitly write $[\![A]\!]_{LA}$ to denote the meaning of A as an LA formula.

For an action A, we define $\Box A$ to be the temporal formula that is true of a behavior iff A is true of all steps of the behavior. In other words, we define the meaning $[\![\Box A]\!]$ of the RTLA formula $\Box A$ by

(3.12) $[\![\Box A]\!](\sigma) \;\triangleq\; \forall\, n \in Nat \,:\, [\![A]\!]_{LA}(\sigma(n) \rightarrow \sigma(n+1))$

Like most logics, RTLA contains the operators of propositional and predicate logic, where they have their standard meanings. For example, $[\![F \wedge G]\!](\sigma)$ equals $[\![F]\!](\sigma) \wedge [\![G]\!](\sigma)$, and $[\![\exists\, i : F]\!](\sigma)$ equals $\exists\, i : [\![F]\!](\sigma)$. As in LA, bound identifiers are called *bound constants* and they act like constants, having the same value in all states of a behavior. Bounded quantification is defined as in ordinary math, with $\forall\, i \in S : F$ equal to $\forall\, i : (i \in S) \Rightarrow F$. In temporal logic, we almost always write such a formula only when S is a constant expression.

It's important to remember that a behavior is *any* cardinal sequence of states. It doesn't have to be a behavior of any particular program. Since any step is the first step of lots of behaviors, it's obvious that if A is an LA formula, then $\models A$ is true when A is viewed as an RTLA formula iff it's true when A is viewed as an LA formula.

Now let's return to the description of program *Increment* in Figure 3.5. It tells us that a behavior σ is a behavior of the program iff (i) the initial predicate *Init* is true of its first state $\sigma(0)$ and (ii) the step predicate *Next* is true for every step $\sigma(n) \rightarrow \sigma(n+1)$ of σ. Condition (i) is expressed by $[\![\mathit{Init}]\!]$, since (3.11) tells us that $[\![\mathit{Init}]\!](\sigma)$ equals $[\![\mathit{Init}]\!]_{LA}(\sigma(0) \rightarrow \sigma(1))$; and since *Init* is a state predicate, it's true of a step iff it's true of the first state of the step. By (3.12), condition (ii) is expressed as $[\![\Box \mathit{Next}]\!]$. Thus (the meaning of) the formula $\mathit{Init} \wedge \Box \mathit{Next}$ is true of a behavior σ iff σ is a behavior of program *Increment*.

Of course, this is true for an arbitrary program. The behaviors that satisfy a program with initial predicate *Init* and next-state action *Next* are described by the simple RTLA formula $\mathit{Init} \wedge \Box \mathit{Next}$. Any program is described by an RTLA formula of this form. As promised, we can write any program as a mathematical formula. It's an RTLA formula rather than a TLA formula, and we'll see that it needs to be modified. But for now, it's close enough to the final TLA formula.

By (3.12), the state predicate *Inv* is true in all states of a behavior iff $\Box \mathit{Inv}$ is true of that behavior. That *Inv* is an invariant of *Increment* means that, for any behavior σ, if σ is a behavior of *Increment* then *Inv* is true in all states of σ. Thus, that *Inv* is an invariant of *Increment* is expressed by this condition:

(3.13) $\models \mathit{Init} \wedge \Box \mathit{Next} \;\Rightarrow\; \Box \mathit{Inv}$

Remember that in (3.10) and (3.13), when *Init*, *Next*, and *Inv* are the formulas defined in Figure 3.5, $\models F$ means that F is true for all interpretations satisfying the assumptions we made about the constants of *Increment*—namely, that *Procs* is a finite set and the implicit assumption that the values of a, b, and *done* are different from one another.

In general, the conditions I1 and I2 for showing that a state predicate *Inv* is an invariant of a program $\mathit{Init} \wedge \Box \mathit{Next}$ are expressed in LA by conditions (3.10). It is an RTLA proof rule that these conditions imply (3.13). When we prove a safety property like (3.13), the major part of the reasoning depends on the definitions of the formulas *Init*, *Next*, and *Inv*. That reasoning is reasoning about actions, which is formalized by LA. The temporal logic reasoning, which is done in RTLA, is trivial. Describing the program with a single formula is elegant. But it is really useful only when verifying liveness properties, which requires nontrivial temporal reasoning.

Because it's often forgotten, it is worth repeating that a state is *any* assignment of values to variables, and a behavior is *any* infinite sequence of states. Even when we are discussing program *Increment*, "state" means any state, including states in which x has the value $\langle \sqrt{2}, 1\,..\,147\rangle$. In (3.13), $\models$ means true for *any* behavior, even behaviors in which the initial value of x is $\langle \sqrt{2}, 1\,..\,147\rangle$. It is true for those behaviors because *Init* equals FALSE for them (unless $\langle \sqrt{2}, 1\,..\,147\rangle$ happens to equal 0, which it might).

3.4.2.2 The Complete RTLA

Simple RTLA suffices for reasoning about safety properties, but not for liveness properties. For example, we want to express the liveness property of program *Increment* that a process p eventually terminates. Termination of p means that $pc(p)$ eventually equals *done* and remains equal to *done* forever. In terms of explicit state numbers, where $pc(n)$ is the value of pc in state number n, this property can be written:

(3.14) $\exists\, j \in \mathbb{N} : \forall\, k \in \mathbb{N} : pc(j+k)(p) = done$

To write it without explicit state numbers, we need full RTLA, in which $\Box$ can be applied to any RTLA formula, not just to an action. To define the meaning of all RTLA formulas, we first define σ^{+n}, for any behavior σ and natural number n, to be the behavior obtained by removing the first n states from the sequence σ. That is, σ^{+n} is the behavior

$$\sigma(n) \to \sigma(n+1) \to \sigma(n+2) \to \cdots$$

so σ^{+n} equals $i \in \mathbb{N} \mapsto \sigma(i+n)$.

For any RTLA formula F, the RTLA formula $\Box F$ is true of a behavior σ iff it is true of the behaviors σ^{+n} for all $n \in \mathbb{N}$. In other words:

(3.15) $[\![\Box F]\!](\sigma) \;\triangleq\; \forall\, n \in Nat : [\![F]\!](\sigma^{+n})$

for any behavior σ. When F is an action A, this is the same definition as (3.12) because the first step $\sigma^{+n}(0) \to \sigma^{+n}(1)$ of σ^{+n} is $\sigma(n) \to \sigma(n+1)$.

Although $\Box$ has a simple definition, temporal formulas can be hard to understand at first. It helps to think of a temporal formula as an assertion about the present and future. The state $\sigma(n)$ of a behavior σ is the state at time n, and the behavior σ^{+n} is the part of the behavior σ that begins at time n. We can then think of $[\![F]\!](\sigma^{+n})$ as asserting that F is true at time n of behavior σ. Thus, $[\![F]\!](\sigma)$ asserts that F is true at time 0 of σ, and $[\![\Box F]\!](\sigma)$ asserts that F is true at all times of σ. The formula $\Box F$ therefore asserts that F is true at all times—that is, F is always true. (Remember that time n is just some instant of time; it is *not* n time units after time 0.)

We now drop the $[\![\ ]\!]$ and think about temporal logic the same way we think about ordinary math, conflating a formula with its meaning. So, we'll think of a temporal formula F as a Boolean-value function on behaviors. However, we will still turn to the formal meaning (3.15) of $\Box$ when it is useful. Sections 3.4.2.3–3.4.2.8 below examine $\Box$ and temporal operators defined in terms of $\Box$. They present quite a few tautologies. Understanding intuitively why those tautologies are true will make you comfortable reading temporal logic formulas and thinking in terms of them.

3.4.2.3 The $\Box$ Operator

We first consider some temporal logic tautologies—theorems about arbitrary temporal formulas. If they are not obvious, rewrite them in terms of the meanings of the formulas. The first tautology asserts the obvious fact that if F is always true, then it is true now:

(3.16) $\models \Box F \Rightarrow F$

The next tautology asserts that $F \land G$ true at all times is equivalent to F true at all times and G true at all times:

(3.17) $\models \Box(F \land G) \ \equiv\ (\Box F) \land (\Box G)$

You should check that (3.17) follows from the definition (3.15) of $\Box$ and the predicate-logic tautology:

$$\models\ (\forall\, n \in S : P \land Q)\ \equiv\ (\forall\, n \in S : P)\ \land\ (\forall\, n \in S : Q)$$

Observe that $\Box(F \lor G)$ and $(\Box F) \lor (\Box G)$ are not equivalent. For example, $\Box(F \lor G)$ is true of a behavior in which F is true only in the initial state and G is false in the initial state and true in all other states. However, neither $\Box F$ nor $\Box G$ is true of that behavior.

We can generalize (3.17) to any conjunction, including an infinite conjunction—that is, quantification over an infinite set of formulas. If F_i is a temporal formula for all $i \in S$, then:

(3.18) $\models\ \Box(\forall\, i \in S\ :\ F_i)\ \equiv\ (\forall\, i \in S\ :\ \Box F_i)$

The next tautology should also be obvious. It asserts that if F implies G is true at all times, then F is true at all times implies G is true at all times:

(3.19) $\models \Box(F \Rightarrow G) \Rightarrow (\Box F \Rightarrow \Box G)$

Perhaps less obvious is this proof rule, which is sort of a converse of (3.16):

(3.20) $\models F$ implies $\models \Box F$

The assertion $\models F$ means that F is true of all behaviors. The assertion $\models \Box F$ asserts that for any behavior, F is true of the part of that behavior starting at any time. But that part of the behavior is itself a behavior, so $\models F$ implies that F is true of it. If this is not obvious to you, then you may be thinking of a behavior as a behavior of some program. A behavior is any infinite sequence of states, so if you remove the first n states of any behavior, you get a behavior.

From (3.19) and (3.20) we easily derive:

(3.21) $\models F \Rightarrow G$ implies $\models \Box F \Rightarrow \Box G$

This rule lies at the heart of much temporal logic reasoning.

We now examine some additional temporal assertions that can be defined using $\Box$.

3.4.2.4 Eventually ($\Diamond$)

The operator $\Diamond$ is defined to be the dual of $\Box$:

(3.22) $\Diamond F \;\triangleq\; \neg\Box\neg F$

Like $\Box$, the operator $\Diamond$ binds more tightly than the operators of propositional logic, so $\Diamond F \wedge G$ is parsed as $(\Diamond F) \wedge G$. To understand $\Diamond$, we derive the meaning $[\![\Diamond F]\!]$ of a formula $\Diamond F$ from (3.15):

$$\begin{array}{rcll}
[\![\Diamond F]\!](\sigma) & \equiv & [\![\neg\Box\neg F]\!](\sigma) & \text{by definition of } \Diamond \\
& \equiv & \neg\,[\![\Box\neg F]\!](\sigma) & \text{by the meaning of } \neg \\
& \equiv & \neg\,\forall\, n \in Nat \,:\, [\![\neg F]\!](\sigma^{+n}) & \text{by (3.15)} \\
& \equiv & \neg\,\forall\, n \in Nat \,:\, \neg\,[\![F]\!](\sigma^{+n}) & \text{by the meaning of } \neg \\
& \equiv & \exists\, n \in Nat \,:\, [\![F]\!](\sigma^{+n}) & \text{by the duality relation (2.21)}
\end{array}$$

Hence, $\Diamond F$ asserts that F is true at some time—either now or in the future. We read $\Diamond$ as *eventually*, where by being eventually true we include the possibility of being true only now. Corresponding to the tautologies (3.16) and (3.17) for $\Box$ are these tautologies for $\Diamond$:

(3.23) $\models F \Rightarrow \Diamond F \qquad \models \Diamond(F \lor G) \;\equiv\; (\Diamond F \lor \Diamond G)$

Make sure you understand why they are true from the meaning of $\Diamond$ as *eventually*. These two tautologies can be derived from (3.16) and (3.17). For example:

(3.24) $(F \Rightarrow \Diamond F) \;\equiv\; (\neg(\neg F) \Rightarrow \neg(\Box\neg F))$ By logic and the definition of $\Diamond$.
$\equiv\; (\Box\neg F \Rightarrow \neg F)$ By propositional logic.

and $\models \Box\neg F \Rightarrow \neg F$ follows from (3.16). You should convince yourself that $\Diamond(F \land G)$ and $(\Diamond F) \land (\Diamond G)$ need not be equivalent. The equivalence of $\Diamond(F \lor G)$ and $\Diamond F \lor \Diamond G$ generalizes to arbitrary disjunctions:

$$\models \Diamond(\exists\, i \in S : F_i) \;\equiv\; (\exists\, i \in S : \Diamond F_i)$$

Here are three tautologies relating $\Diamond$ and $\Box$. The first is obtained by negating $\Diamond F$ and its definition; the second by substituting $\neg F$ for F in that definition; and the third by substituting $\neg F$ for F in the first:

(3.25) $\models \neg\Diamond F \;\equiv\; \Box\neg F \qquad \models \neg\Box F \;\equiv\; \Diamond\neg F \qquad \models \Box F \;\equiv\; \neg\Diamond\neg F$

They should be obvious from thinking of $\Box$ as *always* and $\Diamond$ as *eventually*. The first two tell us that moving $\neg$ over a temporal operator $\Box$ or $\Diamond$ changes $\Box$ to $\Diamond$ and $\Diamond$ to $\Box$. Note the similarity between $\Box/\Diamond$ and $\forall/\exists$, a similarity that arises from the meanings of $\Box$ and $\Diamond$. Another tautology that should also be obvious from thinking of $\Box$ and $\Diamond$ as *always* and *eventually* is:

(3.26) $\models (\Box F \land \Diamond G) \Rightarrow \Diamond(\Box F \land G)$

We can express liveness properties with $\Diamond$. For example, the assertion that some state predicate P is eventually true is a liveness property. The assertion that the program whose formula is F satisfies this property is $\models F \Rightarrow \Diamond P$. Since the assertion that something eventually happens is a liveness property, most of the formulas we write that contain $\Diamond$ express liveness.

The equivalence of $\neg\Diamond P$ and $\Box\neg P$ for any formula P implies the tautology $\models \Diamond P \lor \Box\neg P$. This tautology is central to many liveness proofs. To prove that a program described by formula F satisfies the liveness property $\Diamond P$, we must prove $\models F \Rightarrow \Diamond P$. The tautology $\models \Diamond P \lor \Box\neg P$ and propositional logic imply that we can prove this by proving $\models F \land \Box\neg P \Rightarrow \Diamond P$. This is a proof by contradiction, allowing us to use the additional hypothesis $\Box\neg P$, which should be false. For example, if P is a state predicate, this gives us an invariant $\neg P$ that can be used to prove other invariants.

3.4.2.5 Eventually Always ($\Diamond\Box$)

Recall that termination of process p of program *Increment* means that $pc(p)$ eventually equals *done* and remains forever equal to *done*, a property expressed with explicit state numbers by (3.14). It can be stated as: It's eventually true that $pc(p) = done$ is always true. This is expressed in RTLA as $\Diamond\Box(pc(p) = done)$. You should check that $[\![\Diamond\Box(pc(p) = done)]\!](\sigma)$ is equivalent to (3.14) because $(\sigma^{+j})^{+k}$ equals $\sigma^{+(j+k)}$, so $pc(j + k)(p)$ in (3.14) equals $[\![pc(p)]\!](\sigma^{+(j+k)})$.

We can think of $\Diamond\Box$ as a temporal operator meaning *eventually always.* Convince yourself that this is a tautology:

$$\models \Diamond\Box(F \land G) \;\equiv\; (\Diamond\Box F) \land (\Diamond\Box G)$$

3.4.2.6 Infinitely Often ($\Box\Diamond$)

It should now seem natural to think of $\Box\Diamond F$ as meaning *always eventually* F is true. If you're not used to thinking about infinite sequences, it may not be obvious that *always eventually* is equivalent to *infinitely often.* So, let's prove it.

Theorem 3.1 F is infinitely often true iff it is always eventually true.

Define S_σ to be the set of times at which F is true of a behavior σ.

1. F is infinitely often true of σ iff S_σ is an infinite set.

   PROOF: By definition of *infinitely often.*

2. S_σ is an infinite set iff for every time n, there is a time $m \geq n$ such that $m \in S_\sigma$.

   PROOF: A set of natural numbers is infinite iff it has no largest element.

3. The statement that for every time n there exists a time $m \geq n$ such that $m \in S_\sigma$ is equivalent to the statement that F is always eventually true.

   PROOF: By the definitions of S_σ and *always eventually.*

4. Q.E.D.

   PROOF: Steps 1–3 are of the form A iff B, B iff C, and C iff D, and the theorem asserts A iff D.

It is usually most helpful to think of $\Box\Diamond$ as meaning *infinitely often* rather than *always eventually.* For example, consider the formula $\Box\Diamond(F \lor G)$. It asserts that $F \lor G$ is true infinitely often, which means that F or G is true infinitely often. But F or G is true infinitely often iff at least one of them is true infinitely often. This yields the following tautology:

(3.27) $\models \Box\Diamond(F \lor G) \;\equiv\; (\Box\Diamond F) \lor (\Box\Diamond G)$

The rules for moving $\neg$ over $\Box$ and $\Diamond$ that are implied by the first two tautologies of (3.25) yield the following two tautologies. For example, the first comes from $\neg\Box\Diamond F \;\equiv\; \Diamond\neg\Diamond F \;\equiv\; \Diamond\Box\neg F$.

(3.28) $\models \neg\Box\Diamond F \;\equiv\; \Diamond\Box\neg F \qquad \models \neg\Diamond\Box F \;\equiv\; \Box\Diamond\neg F$

3.4.2.7 The End of the Line

You might expect that we can keep constructing more and more complicated operators like $\Box\Diamond\Diamond\Box\Diamond\Box$ with sequences of $\Box$ and $\Diamond$. We can't. Any such sequence is equivalent to $\Box$, $\Diamond$, $\Box\Diamond$ or $\Diamond\Box$. To see this, first observe that *always always* is the same as *always*. That is, $\Box\Box F$ is equivalent to $\Box F$. That's because $\forall\, i, j \in \mathbb{N} : P(i + j)$ is equivalent to $\forall\, k \in \mathbb{N} : P(k)$, for any P.

Similarly, *eventually eventually* is the same as *eventually*, so $\Diamond\Diamond F$ is equivalent to $\Diamond F$. The equivalence of $\Diamond\Diamond$ and $\Diamond$ also follows from the definition of $\Diamond$ and the equivalence of $\Box\Box$ and $\Box$ by:

$$\Diamond\Diamond F \;\equiv\; \neg\Box\neg\neg\Box\neg F \;\equiv\; \neg\Box\Box\neg F \;\equiv\; \neg\Box\neg F \;\equiv\; \Diamond F$$

So, we can only get a new operator by alternating $\Box$ and $\Diamond$. However, $\Box\Diamond$ and $\Diamond\Box$ is as far as we can go because of the following tautologies:

(3.29) $\models \Diamond\Box\Diamond F \;\equiv\; \Box\Diamond F \qquad \models \Box\Diamond\Box F \;\equiv\; \Diamond\Box F$

The first one is obvious if we read $\Diamond\Box\Diamond$ as *eventually infinitely often*, because F is true at infinitely many times iff it is true at infinitely many times after some time has passed. You can convince yourself that the second is true by realizing that infinitely often F always true is equivalent to F being always true starting at some time. Alternatively, you can show that the first tautology implies the second by figuring out why each of the following equivalences is true:

$$\begin{aligned}\Box\Diamond\Box F \;&\equiv\; \neg\Diamond\neg\neg\Box\neg\neg\Diamond\neg F \;\equiv\; \neg\Diamond\Box\Diamond\neg F \;\equiv\; \neg\Box\Diamond\neg F\\ &\equiv\; \neg\neg\Diamond\neg\neg\Box\neg\neg F \;\equiv\; \Diamond\Box F\end{aligned}$$

3.4.2.8 Leads To ($\leadsto$)

Although there are no more operators to be defined by directly stacking $\Box$ and $\Diamond$, there is another useful temporal operator defined in terms of them: the operator $\leadsto$, read *leads to*, defined by:

$$F \leadsto G \;\triangleq\; \Box(F \Rightarrow \Diamond G)$$

The operator $\leadsto$ is parsed like $\Rightarrow$.

Formula $F \leadsto G$ asserts of a behavior that, whenever F is true, G is true then or later. You should convince yourself that $\leadsto$ is transitive, meaning:

$$\models (F \leadsto G) \land (G \leadsto H) \;\Rightarrow\; (F \leadsto H)$$

Here are two additional tautologies that should be obvious:

(3.30) $\models ((F \lor G) \leadsto H) \;\equiv\; (F \leadsto H) \land (G \leadsto H)$
$\models (F \leadsto G) \land \Box(G \Rightarrow H) \;\Rightarrow\; (F \leadsto H)$

The first of these tautologies generalizes to:

(3.31) $\models ((\exists\, i \in S : F_i) \leadsto H) \;\equiv\; (\forall\, i \in S : (F_i \leadsto H))$

Here are three more tautologies involving $\leadsto$; try to understand why they're true.

(3.32) (a) $\models \Box F \land (F \leadsto G) \;\Rightarrow\; \Box\Diamond G$
(b) $\models (F \leadsto G) \;\equiv\; (F \land \Box\neg G \leadsto G)$
(c) $\models (F \land \Box G \leadsto H) \;\equiv\; (F \land \Box G \leadsto H \land \Box G)$

Here's how I understand them:

(a) $F \leadsto G$ implies that whenever F is true, G is true then or later; and $\Box F$ implies that F is always true. Therefore, $\Box F \land (F \leadsto G)$ implies G is true infinitely often.

(b)
$$\begin{array}{rcll} F \leadsto G & \equiv & \Box(F \Rightarrow \Diamond G) & \text{Definition of } \neg. \\ & \equiv & \Box(F \land \neg\Diamond G \Rightarrow \Diamond G) & \text{Propositional logic.} \\ & \equiv & \Box(F \land \Box\neg G \Rightarrow \Diamond G) & \text{By (3.25).} \\ & \equiv & F \land \Box\neg G \leadsto G & \text{Definition of } \neg. \end{array}$$

(c) $F \land \Box G \leadsto H$ asserts that, for all t, if $F \land \Box G$ is true at time t, then H is true at some time $u \geq t$; and $\Box G$ true at time t implies it is still true at time u, so $H \land \Box G$ is true at time u.

Observe that tautology (b) justifies a proof by contradiction: to prove that F true implies G is eventually true, we can assume that G is never true.

Proving $\leadsto$ properties is at the heart of liveness proofs. For example, here's how we prove termination of Euclid's algorithm, discussed in Section 1.5. The algorithm terminates because while $x \neq y$ is true, the sum of x and y keeps decreasing, which can't continue forever because the algorithm satisfies the invariant that x and y are positive integers. Therefore, eventually $x = y$ and the algorithm terminates. This argument is formalized in RTLA as follows.

To prove termination, we must prove that every behavior of the algorithm satisfies $\Diamond(x = y)$. The proof uses this tautology, which follows from the meanings of the operators $\Box$, $\leadsto$, and $'$:

(3.33) $\models \Box(P \Rightarrow Q') \Rightarrow (P \leadsto Q)$

The tautology $\models \Diamond P \lor \Box\neg P$ allows us to prove $\Diamond(x = y)$ by assuming that a behavior of the algorithm satisfies $\Box(x \neq y)$ and obtaining a contradiction. Let R_i be the state predicate $x + y \leq i$. We prove that $\Box(x \neq y)$ and the invariant that x and y are positive integers imply that, for all $i > 0$, the program satisfies $\Box(R_i \Rightarrow (R_{i-1})')$. By (3.33), this implies $R_i \leadsto R_{i-1}$. By the transitivity of $\leadsto$ and mathematical induction, this implies $R_{M+N} \leadsto R_0$. Since the program implies that R_{M+N} is true in the initial state, this implies that $\Diamond R_0$ is true, contradicting the invariant that x and y are always positive.

3.4.2.9 Warning

Although elegant and useful, temporal logic is weird. It's not ordinary math. In ordinary math, any operator *Op* we can define satisfies the condition, sometimes called *substitutivity*, that the value of an expression $Op(e_1, \ldots, e_n)$ is unchanged if we replace any e_i by an expression equal to e_i. If *Op* takes a single argument, substitutivity means that

(3.34) $\models (exp_1 = exp_2) \Rightarrow (Op(exp_1) = Op(exp_2))$

is true for any expressions exp_1 and exp_2. For example, (3.34) is true for the operator *Tail*. However, the temporal operator $\Box$ is not substitutive. For example let exp_1 and exp_2 be the state predicates $x = 0$ and $y = 0$, respectively; and let σ be a behavior such that for each $j \in \mathbb{N}$, the state $\sigma(j)$ assigns the value 0 to x and the value j to y. Then exp_1 and exp_2 both equal TRUE for σ because they are both true for $\sigma(0)$. Formula

$\Box exp_1$ is true for σ because $x = 0$ is true in all its states, but $\Box exp_2$ is false for σ because $y = 0$ is true only in the first state $\sigma(0)$. Hence, the value of the formula $(exp_1 = exp_2) \Rightarrow (\Box exp_1 = \Box exp_2)$ for this behavior is $(\text{TRUE} = \text{TRUE}) \Rightarrow (\text{TRUE} = \text{FALSE})$, which equals FALSE. The operator $'$ (prime) is similarly not substitutive, so it too is weird. This weirdness affects all temporal logics and makes temporal logic reasoning tricky.

3.5 TLA

3.5.1 The Problem

There is something terribly wrong with our RTLA descriptions of abstract programs, because there is something terribly wrong with the descriptions like the one in Figure 3.4 that we wrote using explicit step numbers. To see why, let's return to the discussion in Section 3.1 of how astronomers describe a planet orbiting a star. As explained there, the mathematical description of the orbiting planet is best thought of as describing a universe containing the planet, saying nothing about what else is or is not in the universe. In particular, that description applies just as well to a universe in which there is a spacecraft close to the star that orbits it very fast—perhaps going around the star 60 times every time the planet goes around it once. Since the spacecraft is too small to affect the motion of the planet, we would obtain a description of the system composed of the planet and the spacecraft by adding (conjoining) a description of the spacecraft's motion to the description of the planet's motion. The description of the planet's motion remains an accurate description of that planet in the presence of the spacecraft. It would be crazy if we had to write different formulas to describe the planet because of the spacecraft that has no effect on it.

Now consider the descriptions of abstract programs we've been writing. In particular, consider an RTLA formula *HM* describing how the values of the hour and minute displays of a 24-hour clock change. Using the variables *hr* and *min* to describe the current hour and minute being displayed, we might define *HM* to equal $Init \land \Box Next$, where:

$$\begin{aligned}(3.35)\quad Init &\triangleq (hr = 0) \land (min = 0)\\ Next &\triangleq \land\ min' = (min + 1) \,\%\, 60\\ &\ \land\ hr' = \text{IF } min = 59 \text{ THEN } (hr + 1) \,\%\, 24 \text{ ELSE } hr\end{aligned}$$

But suppose that the clock also displays seconds. The RTLA formula *HMS* that also describes the second display might use a variable *sec* to describe that display. A behavior σ allowed by *HMS* would not be allowed by *HM* because *HM* requires every step to change the value of *min*, while σ must change the value of *sec* in every step and the value of *min* in only every 60th step.

It is just as crazy for an abstract program describing an hour-minute clock not to describe a clock that also displays seconds as it is for a description of a planet's motion no longer to describe that motion because of a spacecraft that doesn't affect the planet. It means that anything we've said about the hour and minute display might be invalid if there's also a second display. And it doesn't matter if the minute display is on a digital clock on my desk and the second display is on a phone in my pocket. More generally, it means if we've proved things about completely separate digital devices and we look at those two devices at the same time, nothing we've proved about them remains true unless those devices are somehow synchronized to run in lock step. The more you think about it, the crazier it seems.

3.5.2 The Solution

To figure out how to fix this problem, let's first see where we went wrong. It happened in Section 3.2.1, when we went from a sequence t_0, t_1, t_2, ... of times to a sequence 0, 1, 2, ... of state numbers. We were writing a description of a particular system. But math and science don't describe a system; they describe a universe containing that system. And that universe can contain many systems. A different system might lead to a different sequence u_0, u_1, u_2, ... of times, with only t_0 and u_0 equal. Our error was converting two possibly different times t_i and u_i into the same state number. The result was that when we thought we were writing a description of a particular system, we were actually writing a description of a universe in which the values of all variables, including ones describing other systems, could change only when the variables of that particular system changed.

You might think that because the error occurred when we were throwing away times, we need to represent the time at which a state holds, not just a state number. Fortunately, there is a simpler solution. It's the one we used to eliminate finite behaviors and consider only infinite behaviors. We observed that we could do that by adding *stuttering steps* at the end of a finite sequence of states—steps that just repeat the previous state of the program. Eliminating finite behaviors was not simply a matter of convenience. The real reason to do it was to eliminate one source of craziness. Since a behavior is not just a behavior of a particular program but a behavior of the entire universe, a finite behavior is one in which everything in the universe stops changing. The description of a halting program execution as a finite behavior therefore asserts that the entire universe halts when the program does. Those infinitely many stuttering steps, in which the value of no variable of the program changes, allows other programs' variables to keep changing.

We can add those stuttering steps because of the observation that the conversion from times to state numbers requires that a program variable be allowed to change only at time t_i for some i. It does not require that any variable does change at that time. The mistake was writing descriptions that, until the program halts, requires some variable to change value at each time t_i. Instead, we should have added to the sequence of times t_i times at which no program variable changes. Adding such a time adds a step in which other variables describing other programs can change while the program's variables remain unchanged. Thus, if the description allows a behavior σ, then it should allow the behavior obtained by inserting stuttering steps of the program in σ. This is easy to do. For the description of the hour/minute display, we just change the definition of HM to

$$HM \;\triangleq\; Init \wedge \Box(Next \vee ((hr' = hr) \wedge (min' = min)))$$

We can write this formula more compactly as

$$(3.36)\quad HM \;\triangleq\; Init \wedge \Box(Next \vee (\langle hr, min\rangle' = \langle hr, min\rangle))$$

because $\langle hr, min\rangle'$ equals $\langle hr', min'\rangle$, and two tuples are equal iff their corresponding components are equal.

We can similarly fix every other example we've seen so far by changing the next-state action $Next$ in its RTLA description to $Next \vee (v' = v)$, where v is the tuple of all variables that appear in the RTLA formula. Since this will have to be done all the time, we abbreviate $A \vee (v' = v)$ as $[A]_v$ for any action A and state expression v.

We can add stuttering steps to a pseudocode description of an algorithm by adding a separate process that just takes stuttering steps. However, we won't bother to do this. We will just consider all pseudocode to allow stuttering steps.

When HM is defined by (3.36), if HMS is true of a behavior then HM is also true of the behavior. This remains true when HMS is modified to allow stuttering steps. Thus, HMS implements HM, and $\models HMS \Rightarrow HM$ is true. Implementation is implication. How elegant!

There is an apparent problem with formula HM of (3.36). It allows behaviors in which the program takes a finite number of steps (possibly zero steps) and then takes nothing but stuttering steps. In other words, it allows behaviors in which the clock stops. Most computer scientists will say that we should never allow behaviors in which an abstract program stops when it is possible for it to continue executing. This is because they are used to thinking about traditional programs. In many cases, we don't want to require a concurrent abstract program to do something just because it can.

Never stopping is a liveness property. Taking only steps satisfying $[Next]_v$ is a safety property. My experience has taught me that we should describe safety properties separately from liveness properties because we reason about them differently and we should think about them differently. Formula *HM* describes the safety property that the hour-minute clock should satisfy. We will see later how we conjoin a liveness property to *HM* if we want to require the clock to run forever. It is a feature not a problem that this definition of *HM* asserts only what the clock *may* do and not what it *must* do.

In general, the safety property of an abstract program is written in the form $Init \wedge \Box[Next]_v$, where *Init* is the initial predicate and $[Next]_v$ is the next-state action. The formula $\Box[Next]_v$ always allows stuttering steps because $[Next]_v$ has the form $\ldots \vee (v' - v)$, and $v' = v$ allows stuttering steps. However, $v' = v$ allows lots of non-stuttering steps. In particular, it allows steps in which any variable that does not appear in v can have any values in the two states of the step. To describe an abstract program, the state expression v in $\Box[Next]_v$ must ensure that $v' = v$ allows only steps that do not change any of the program's variables. Therefore, unless stated otherwise, in a formula of the form $\Box[Next]_v$ where *Next* is the next-state action of a program, the subscript v is assumed to be the tuple of all program variables. (However, that subscript need not be called v.)

3.5.3 Stuttering Insensitivity

We have seen that the safety property of an abstract program should have the form $Init \wedge \Box[Next]_v$, so it allows stuttering steps. But what can we say in general about formulas for describing systems or abstract programs?

Stuttering steps are created by adding extra times t_i at which we report the values of a program's variables. A stuttering step does not represent the program doing anything. It's just a mathematical way to allow the descriptions of the universe with which we describe different abstract programs to be made consistent with one another. Therefore, any assertion we make about a behavior of an abstract program should not depend on whether we add or remove steps that leave the program's variables unchanged. Since the assertion depends only on the values a behavior assigns to the program's variables, this condition is satisfied iff the assertion does not depend on whether we add or remove steps that leave *all* variables unchanged. We've used the term *stuttering step* to mean a step that leaves a program's variables unchanged. We will now call such a step a stuttering step *of the program.* We define a stuttering step to be a step that leaves *all* variables unchanged.

A sensible predicate F on behaviors should satisfy the condition that the value of $[\![F]\!](\sigma)$ is not changed by adding stuttering steps to, or removing them from, a behavior σ. This means that the value of $[\![F]\!](\sigma)$ is not changed even if an infinite number of stuttering steps are added and an infinite number removed. (However, the behavior must still be infinite, so if σ ends in an infinite number of stuttering steps, those steps can't be removed.) A predicate on behaviors satisfying this condition for all behaviors σ is called *stuttering insensitive*, or SI for short. When describing abstract programs or the properties they satisfy, we should use only SI predicates on behaviors.

To define SI precisely, we first define $\natural(\sigma)$ to be the behavior obtained by removing from the behavior σ all stuttering steps except those belonging to an infinite sequence of stuttering steps at the end. Recall that $Tail(\sigma)$ is the behavior obtained from σ by removing its first state and $\circ$ is concatenation of sequences. For a state s, let's define $(s \rightarrow)$ to equal $i \in \{0\} \mapsto s$, the cardinal sequence of length 1 whose single item is s. Here is the recursive definition of $\natural$.

$$\begin{array}{lll}(3.37) \ \natural(\sigma) & \triangleq & \text{IF } \forall\, i \in \mathbb{N} : \sigma(i) = \sigma(0) \\ & & \quad \text{THEN } \sigma \\ & & \quad \text{ELSE } \text{ IF } \sigma(0) = \sigma(1) \text{ THEN } \natural(Tail(\sigma)) \\ & & \qquad\qquad\qquad\qquad\qquad\ \text{ELSE } (\sigma(0) \rightarrow) \circ \natural(Tail(\sigma))\end{array}$$

A predicate on behaviors is defined to be SI iff, for any behavior σ, the predicate is true of σ iff it is true of $\natural(\sigma)$. SI is a semantic condition—that is, a condition on the meanings of formulas. Since we are conflating formulas and their meanings, saying that a formula F is SI means that $[\![F]\!]$ is SI.

We have been using the term *property* informally to mean some condition on the behaviors of a system or abstract program. We now define it to mean an SI predicate on behaviors. *Behavior predicate* still means any predicate on behaviors, not just SI ones.

3.5.4 The Definition of TLA

We now define TLA to be a language that is a sublanguage of RTLA in which every formula is a property—that is, an SI formula. Defining a language means giving syntactic rules for a formula to belong to the language.

We begin by defining a state predicate to be a TLA formula. Viewed as a temporal formula, a state predicate is SI because it depends only on the first state of a behavior, which isn't changed by adding or removing stuttering steps.

The operators of propositional logic applied to SI formulas produce SI formulas. For example, if adding or removing stuttering steps doesn't change whether formulas F and G satisfy a behavior, then they don't change whether $F \land G$ satisfies the behavior. So we let TLA include all formulas obtained by applying propositional logic operators to TLA formulas. Similarly, we can let TLA include all formulas obtained by applying unbounded quantifiers and quantifiers bounded by a state expression to a TLA formula. For example, $\exists\, x \in S : F$ is a TLA formula if F is one and S is a state expression.

It's easy to find actions A that, when viewed as a temporal formula, are not SI. For example, $x' \neq x$ is not SI because if σ is a behavior that satisfies $x' \neq x$, then the behavior obtained by adding a stuttering step to the beginning of σ doesn't satisfy it. However, the formula $\Box[A]_v$ is SI for any action A and state expression v, so we let TLA contain all such formulas.

The formula $\Box[A]_v$ asserts that an action, namely one of the form $[A]_v$, is true of all steps of a behavior. For reasoning about liveness, we will need to assert that an action is eventually true in some step of a behavior. The formula $\Diamond A$ is not SI for an arbitrary action A because if A is true on some stuttering step, then $\Diamond A$ might be false on a behavior σ and true on a behavior obtained by adding such a stuttering step to σ. However, if A does not allow stuttering steps, then $\Diamond A$ is SI. In particular, the formula $\Diamond(A \land (v' \neq v))$ is SI for any state expression v. We define $\langle A \rangle_v$ to equal $A \land (v' \neq v)$; and we let TLA contain all formulas $\Diamond\langle A \rangle_v$, for any action A and state expression v.

You can convince yourself that adding or removing stuttering steps from a behavior doesn't change whether it satisfies $\Diamond\langle A \rangle_v$. However, since $\neg\Box[\neg A]_v$ is SI, the following theorem shows that $\Diamond\langle A \rangle_v$ is SI:

$$\text{(3.38)} \quad \models \Diamond\langle A \rangle_v \;\equiv\; \neg\Box[\neg A]_v$$

This equivalence follows from the definition of $\Diamond$ and the following assertion, which can be proved by propositional logic from the definitions of $[\ldots]_v$ and $\langle \ldots \rangle_v$:

$$\models \langle A \rangle_v \;\equiv\; \neg[\neg A]_v$$

From (3.38) we see that $\neg\Diamond\langle A\rangle_v$ is equivalent to $\Box[\neg A]_v$. This means that an $\langle A\rangle_v$ step never occurs in a behavior iff every step of the behavior is a $[\neg A]_v$ step. This fact is used in proofs by contradiction of formulas of the form $\Diamond\langle A\rangle_v$.

The only temporal operator we have defined besides $\Box$ and $\Diamond$ is $\leadsto$. We define $F \leadsto \Diamond\langle A\rangle_v$ to be a TLA formula if F is one, A is an action, and v is a state expression. This formula is SI, since expanding the definition of $\leadsto$ in it produces what we have already defined to be a TLA formula.

Combining all this, we see that a TLA formula is one of the following:

- A state predicate.
- Obtained by applying propositional logic operators to TLA formulas.
- $\forall\exists\, c : F$ or $\forall\exists\, c \in S : F$ where $\forall\exists$ is $\forall$ or $\exists$, for a constant c, a TLA formula F, and a state expression S.
- $\Box[A]_v$, $\Diamond\langle A\rangle_v$, or $F \leadsto \Diamond\langle A\rangle_v$, for an action A, a state expression v, and a TLA formula F.

Abstract programs and the properties they satisfy should be TLA formulas. However, we can use RTLA proof rules and even RTLA formulas when reasoning about TLA formulas. For example, we can prove that *Inv* is an invariant of $Init \land \Box[Next]_v$ by substituting $[Next]_v$ for *Next* in the RTLA proof rule that (3.10) implies (3.13). This yields the following rule:

$$\begin{array}{l} \models (Init \Rightarrow Inv) \;\land\; (Inv \land [Next]_v \Rightarrow Inv') \\ \qquad \text{implies} \quad \models Init \land \Box[Next]_v \Rightarrow \Box Inv \end{array}$$

In this rule, the first $\models$ means validity in LA while the second $\models$ means validity in TLA. A feature of TLA is that as much reasoning as possible is done in LA, which becomes ordinary mathematical reasoning when the necessary definitions are expanded and primes are distributed across operators so only variables are primed.

Chapter 4

Safety, Liveness, and Fairness

4.1 Safety and Liveness

4.1.1 Definitions

Safety and liveness properties have been described intuitively as specifying what the program is allowed to do and what it must do. To define them precisely, we begin by observing that they have these characteristics:

Safety If a behavior doesn't satisfy a safety property, then we can point to the place in the behavior where it violates the property. For example, if a behavior doesn't satisfy an invariance property, it violates the property in the first state in which the invariant is false.

Liveness We have to look at an entire infinite behavior to see that it doesn't satisfy a liveness property. For example, we can't see that the property *x eventually equals 42* is violated by looking at a finite part of the behavior.[1]

This characterization was turned into precise definitions of safety and liveness for arbitrary behavior predicates by Alpern and Schneider [3]. Since we're interested only in properties, we will use a somewhat simpler definition of safety. But first, we need a few preliminary definitions.

We call a finite, nonempty cardinal sequence of states a *finite behavior*. (A *behavior*, without the adjective *finite*, still means an infinite cardinal

[1]Remember that a behavior means any infinite sequence of states, not just one that satisfies some program. If we know that a behavior satisfies the program, we can often tell that $\diamond(x = 42)$ is false by looking at the behavior predicate that describes the program, without looking at the behavior at all.

sequence of states.) We'll write a finite behavior ρ as $\rho(0) \rightarrow \cdots \rightarrow \rho(n)$. A nonempty finite prefix of a behavior is a finite behavior. We define the *completion* $\rho^{\uparrow}$ of a finite behavior to be the behavior obtained by repeating the last state of ρ infinitely many times—that is, adding infinitely many stuttering steps. A finite behavior ρ is defined to satisfy a behavior predicate iff its completion $\rho^{\uparrow}$ satisfies it. We can now precisely define safety and liveness.

Safety A property F is a safety predicate iff it satisfies the following condition: A behavior satisfies F iff every nonempty finite prefix of the behavior satisfies F.

Liveness A property F is a liveness property iff every finite behavior is the prefix of a behavior that satisfies F.

A state predicate is a safety property because it is satisfied by a behavior iff the state predicate is true on the initial state, and a behavior and all its nonempty prefixes have the same initial state. The formula $\Box[A]_v$ is a safety property for any action A and state expression v; here is the proof that every nonempty finite prefix of a behavior σ satisfies $\Box[A]_v$ iff σ satisfies $\Box[A]_v$.

1. ASSUME: Every nonempty finite prefix of σ satisfies $\Box[A]_v$.
   PROVE: σ satisfies $\Box[A]_v$

   PROOF: σ satisfies $\Box[A]_v$ iff every step of σ satisfies $[A]_v$, and every step of σ is a step of some finite prefix of σ, so it satisfies $\Box[A]_v$ by the assumption.

2. ASSUME: σ satisfies $\Box[A]_v$
   PROVE: Every nonempty finite prefix of σ satisfies $\Box[A]_v$.

   PROOF: Every step of a nonempty finite prefix of σ is either a step of σ, so it satisfies $[A]_v$ by the assumption, or it is a stuttering step, which satisfies $[A]_v$ by definition of $[\ldots]_v$.

3. Q.E.D.

   PROOF: Obvious, by steps 1 and 2.

It also follows easily from the definition of safety that the conjunction of safety properties is a safety property. Therefore, as expected, the formula $\mathit{Init} \land \Box[\mathit{Next}]_v$ that we have been calling the safety property of a program is indeed a safety property.

The property that asserts that a program halts is a liveness property. That property is true of a behavior σ iff σ ends with infinitely many steps that leave the program's variables unchanged. It's a liveness property because every finite behavior ρ is a prefix of its completion $\rho^{\uparrow}$, which satisfies the property.

Safety and liveness are conditions on properties, which are SI behavior predicates. When we say that a TLA formula $\Box[A]_v$ is a safety property, we are conflating the formula with its meaning. We should remember that it's actually $[\![\Box[A]_v]\!]$ that is the safety property.

4.1.2 A Completeness Theorem

TLA is quite simple, adding only the two operators $'$ (prime) and $\Box$ to ordinary math. In theory, this simplicity makes it quite inexpressive. For example, here is a property F_{12} that neither TLA nor RTLA can express: the value of x must equal 1 before it can equal 2. It's expressed in terms of explicit states as:

$$(4.1)\quad F_{12} \;\triangleq\; \forall\, j \in \mathbb{N} : (x(j) = 2) \;\Rightarrow\; \exists\, k \in \mathbb{N} : (k < j) \wedge (x(k) = 1)$$

This behavior predicate is a property because it's SI; adding or removing stuttering steps doesn't affect whether a behavior satisfies it. Property F_{12} is a safety property because it's not satisfied by a behavior σ iff there's a point in σ at which it's violated—namely, a state $\sigma(j)$ in which $x = 2$ and $x \neq 1$ in all the states $\sigma(0), \ldots, \sigma(j-1)$.

If TLA is inexpressive, how can we describe programs with it? The answer is, by using variables. We can express F_{12} as an abstract program described by a TLA formula S_{12} if we add a Boolean-valued variable, let's call it y, whose value is TRUE iff x equals 1 or has previously equaled 1. We let the initial predicate *Init* of S_{12} assert that $x \neq 2$ and that $y = \text{TRUE}$ iff $x = 1$. The next-state relation *Next* allows x' to equal 2 only if $y = \text{TRUE}$, and it sets y to TRUE if $x = 1$. Here are the definitions:

$$(4.2)\quad \begin{aligned} S_{12} &\triangleq \mathit{Init} \wedge \Box[\mathit{Next}]_{\langle x,y \rangle} \\ \mathit{Init} &\triangleq (x \neq 2) \wedge (y = (x = 1)) \\ \mathit{Next} &\triangleq \wedge\ (x' = 2) \Rightarrow y \\ &\ \wedge\ y' = (y \vee (x = 1)) \end{aligned}$$

It's not obvious in what sense formula S_{12} expresses property F_{12}, since S_{12} contains the variables x and y while F_{12} describes only the values of x. Intuitively, S_{12} makes the same assertion as F_{12} if we ignore the value of y. Section 6.1 describes a TLA operator $\boldsymbol{\exists}$ such that $\boldsymbol{\exists}\, y : S_{12}$ means S_{12} *if we ignore the value of* y. We'll then see that $[\![\boldsymbol{\exists}\, y : S_{12}]\!]$ equals F_{12}. However, there's no need to introduce $\exists$ here. The relevant condition that S_{12} satisfies is that if G is any TLA formula that does not contain the variable y, then

$$\models F_{12} \Rightarrow [\![G]\!] \quad \text{iff} \quad \models S_{12} \Rightarrow G$$

The idea of adding a variable to express a property works in general. We state it now only for safety properties. We can't express *every* safety property as a TLA formula. A formula is a finite string of finitely many symbols, and there are only a countable number of such strings; but there are uncountably many safety properties. (For example, there are uncountably many real numbers, so there are uncountably many properties asserting that the initial value of x is a particular real number.) What we can show is that any safety property (which is a predicate on behaviors) that can be described by a mathematical formula—that is, by a formula of ZF—can be expressed as a TLA formula. We do that by showing that if the mapping F from behaviors to Boolean values is a safety property, then we can use F to write a TLA formula that describes it the way S_{12} describes F_{12} in our example. One condition satisfied by a mathematical formula is that it contains only a finite number of variables, and its value depends only on the values of those variables. Remember that we are assuming that the language LA for writing actions contains all the operators of ZF, including the operators on sequences described in Section 2.3.2.

The theorem is expressed with the convention of letting a boldface identifier like $\mathbf{x}$ be the list $x_1, \ldots, x_n$ of subscripted non-bold versions of the identifier, for some n. Thus, $\langle \mathbf{x} \rangle$ is the tuple of those identifiers. The theorem is a special case of Theorem 4.8 in Section 4.2.7 below, so the proof is omitted.

Theorem 4.1 Let $\mathbf{x}$ be the list $x_1, \ldots, x_n$ of variables and let F be a safety property such that $F(\sigma)$ depends only on the values of the variables $\mathbf{x}$ in a behavior σ. There exists a formula S equal to $\mathit{Init} \wedge \Box[\mathit{Next}]_{\langle \mathbf{x}, y \rangle}$, where Init and Next are defined in terms of F, y is a variable not among the variables $\mathbf{x}$, and the variables of S are $\mathbf{x}$ and y, such that $\models F \Rightarrow G$ iff $\models [\![S]\!] \Rightarrow G$, for any property G.

This theorem is a completeness result, showing that TLA can express as an abstract program any safety property that can be expressed semantically. While this shows that there is no fundamental lack of expressiveness in TLA, it is of little practical significance. The proof assumes a description of the property F and uses it to write F as an abstract program. If there were a better way to describe properties mathematically than with abstract programs, we should use it. There are other temporal logics that can express the simple property F_{12} with a formula that's easier to understand than S_{12}. However, S_{12} is not hard to understand, and abstract programs are the only practical way I know to express all the properties of concrete concurrent programs that we need to describe.[2]

4.1.3 The Operator $\mathcal{C}$

We now define the operator $\mathcal{C}$ so that $\mathcal{C}(F)$ is the strongest safety property implied by F, for any property F. Remember that property G stronger than property H means every behavior satisfying G satisfies H—that is, $\models G \Rightarrow H$. The operator $\mathcal{C}$ is not part of the TLA language; we do not use it to write abstract programs. What we do is show that under certain conditions some G equals $\mathcal{C}(F)$ for some other program F. It would be better to write that $[\![G]\!]$ equals $\mathcal{C}([\![F]\!])$, but we won't bother because we regularly conflate a formula and its meaning.

If we want $\mathcal{C}(F)$ to be the strongest safety property implied by F, it should be satisfied by the behaviors satisfying F plus the fewest additional behaviors needed to make it a safety property. The appropriate definition is: A behavior σ satisfies $\mathcal{C}(F)$ iff every finite prefix of σ is a prefix of a behavior that satisfies F. The following theorem shows that this is the correct definition. Its proof is in the Appendix.

Theorem 4.2 If F is a property, then $\mathcal{C}(F)$ is a safety property such that $\models F \Rightarrow \mathcal{C}(F)$ and, for any safety property G, if $\models F \Rightarrow G$ then $\models \mathcal{C}(F) \Rightarrow G$.

Alpern and Schneider proved that every property is the conjunction of a safety property and a liveness property.[3] They actually proved this stronger result, whose proof is in the Appendix.

Theorem 4.3 Every property F is equivalent to $\mathcal{C}(F) \wedge L$ for a liveness property L.

We have been describing abstract programs by formulas of the form $\mathit{Init} \wedge \Box[\mathit{Next}]_v$, which are safety properties. As we've observed, like any safety property, this formula allows behaviors that halt at any point in the behavior. We usually don't want to allow such behaviors, so we must conjoin a liveness property to this formula to describe most abstract programs. For

[2]Remember that *property* means predicate on behaviors. There are many conditions we want programs to satisfy besides properties.

[3]Their result was stated for arbitrary behavior predicates, not properties.

example, we can rule out behaviors of program *Increment* that don't halt prematurely with the liveness property

$$\forall\, p \in \mathit{Procs} : \Diamond(\mathit{pc}[p] = \mathit{done})$$

However, we'll see later why that's not a good liveness property to use.

There's another method of describing safety and liveness that helps me understand them intuitively. It's based on topology. The method and the necessary topology are explained in Appendix Section A.5.

4.1.4 What Good is Liveness?

Safety predicates constrain the finite behavior of a system. They describe what must be true of finite prefixes of a behavior. Liveness properties say nothing about finite prefixes; they describe what must be true if the system runs forever. Since we don't live forever, why should be care about liveness properties?

Liveness is useless in theory but useful in practice. Consider the liveness property required of a traditional program: it eventually terminates. In theory, that's useless because it might not terminate in a billion years. In practice, proving that a program will terminate within a given amount of time isn't easy. Proving that it eventually terminates is easier, and it is useful because the program is certainly not going to terminate soon enough if it never does. But proving liveness provides more than that. Understanding why a program eventually terminates requires understanding what it must do in order to finish. That understanding helps you decide if it will terminate soon enough. This applies to other liveness properties as well.

Using a model checker doesn't give you the understanding that you get from writing a proof. However, checking liveness properties is a good way to detect errors—both in the program you intended to write and in what you actually wrote. A program that does nothing satisfies most safety properties, and an error in translating your intention into mathematics might disallow behaviors in which the program fails to satisfy a safety property. Checking that the program satisfies liveness properties that it should can catch such errors, as well as errors in the program you wanted to write.

4.2 Fairness

Expressing mathematically the way computer scientists and engineers described their algorithms and programs led us to describe the safety property satisfied by an abstract program with the formula $Init \land \Box[Next]_v$, where v is the tuple of all the program's variables. We must conjoin to that formula another formula to describe the program's liveness property. To see how this should be done, we first examine how scientists and engineers expressed liveness.

4.2.1 Traditional Programs and the Enabled Operator $\mathbb{E}$

We start with traditional programs. It was assumed, usually without needing to be stated explicitly, that a program kept executing statements until it terminated. If termination is expressed by $pc = done$, then this assumption can be stated as the requirement that when $pc \neq done$, a $Next$ step must eventually occur. This is expressed by the TLA formula

(4.3) $(pc \neq done) \leadsto \langle Next \rangle_v$

The $\langle \ldots \rangle_v$ is a bit of a nuisance, but it's required by TLA to prevent a liveness property from being satisfied by a stuttering step, which would make no sense. Usually, the next-state action $Next$ does not permit stuttering steps, so (4.3) is equivalent to the RTLA formula $(pc \neq done) \leadsto Next$.

Formula (4.3) assumes a particular way of expressing termination. In general, termination of a traditional program means it's no longer possible for the program to take steps. We should replace (4.3) by $E \leadsto \langle Next \rangle_v$, where E is a state predicate that is true in a state iff it's possible to take a $\langle Next \rangle_v$ step in that state. We write that state predicate E as $\mathbb{E}\langle Next \rangle_v$, where $\mathbb{E}$ is read *enabled.*

In general, for any action A, we define $\mathbb{E}(A)$ to be the state predicate that is true in a state s iff there exists a state t such that $s \to t$ is an A step. In other words, $\mathbb{E}$ is an LA operator, where for any action A the state predicate $\mathbb{E}(A)$ is defined by:

$$[\![\mathbb{E}(A)]\!](s) \;\triangleq\; \exists\, t : [\![A]\!](s \to t)$$

In the common case when A has the form $\langle B \rangle_v$, we omit the parentheses and write simply $\mathbb{E}\langle B \rangle_v$. The liveness property assumed of a traditional program whose safety property is described by the formula $Init \land \Box[Next]_v$ is $\mathbb{E}\langle Next \rangle_v \leadsto \langle Next \rangle_v$.

4.2.2 Concurrent Programs

In traditional programs, when the program hasn't terminated there is just one program statement that can be executed. In multiprocess programs, it is usually possible for there to be multiple statements that can be executed, each in a different process. Moreover, a process can stop not just because it has terminated, but because it is waiting for another process to do something. The liveness property $\mathbb{E}\langle \mathit{Next} \rangle_v \leadsto \langle \mathit{Next} \rangle_v$ ensures that the program keeps executing statements as long as some process hasn't halted. It is satisfied if one process keeps executing statements. It allows other processes to halt, even if they could keep executing statements. Those other processes are said to be *starved*.

It was generally accepted that processes should be treated "fairly". Multiprocess programs were usually executed on computers having fewer processors than there were processes—for many years, usually just a single processor. It was sometimes proposed that fairness should guarantee the stronger condition that each process get a fair share of processor time. However, it came to be generally accepted that fairness should not specify how long (in terms of program steps) a process that can execute a statement might wait before executing it. Therefore, fairness came to mean simply that no process should be starved.

In a program with a set *Procs* of processes, the next-state action is defined by

$$\mathit{Next} \;\triangleq\; \exists\, p \in \mathit{Procs} : \mathit{PNext}(p)$$

where $\mathit{PNext}(p)$ is the next-state action of process p. The obvious generalization of the liveness requirement for a traditional program suggests that fairness for all the processes in a multiprocess program should mean:

$$\text{(4.4)}\quad \forall\, p \in \mathit{Procs} : \mathbb{E}\langle \mathit{PNext}(p) \rangle_v \leadsto \langle \mathit{PNext}(p) \rangle_v$$

However, this is *not* the way fairness should be expressed, and it is not an appropriate liveness property for multiprocess programs. To see why, we consider mutual exclusion algorithms.

4.2.2.1 Mutual Exclusion

The concept of fairness in a concurrent program first appeared (though not by that name) in Edsger Dijkstra's seminal 1965 paper that launched the study of concurrent algorithms [9]. That paper defined mutual exclusion and presented the first algorithm that implemented it.

In mutual exclusion, we assume a set of processes that each alternately executes two sections of code called the *noncritical* and *critical* sections. A mutual exclusion algorithm must ensure that no two processes can be executing their critical sections at the same time. For example, the processes may

```
variables ... ;                          global variables
process p ∈ Procs
   variables pc = ncs, ... ;             process-local variables
   while TRUE do
     ncs:  skip ;                        noncritical section
     wait:   ⋮                           waiting section
     cs:   skip ;                        critical section
     exit:   ⋮                           exiting section
   end while
end process
```

Figure 4.1: The outline of a mutual exclusion algorithm.

occasionally print output on the same printer, and two processes printing at the same time would produce an unreadable mixture of the two outputs. To prevent that, the processes execute a mutual exclusion algorithm, and a process prints only when in its critical section.

The outline of a mutual exclusion algorithm is shown in Figure 4.1, where *Procs* is the set of processes. We don't care what the processes do in their noncritical and critical sections, so we represent them by atomic **skip** statements labeled *ncs* and *cs* that do nothing when executed except change the value of *pc*. The nontrivial part of the algorithm consists of the two sections of code, the *waiting* and *exiting* sections, that begin with the labels *wait* and *exit*. Each of those sections can contain multiple labeled statements, using variables declared in the two **variables** statements.

The safety property that a mutual exclusion algorithm must satisfy is that no two processes are executing their critical sections at the same time—meaning that $pc(p)$ and $pc(q)$ cannot both equal *cs* for two different processes p and q. This is an invariance property. A cute way of expressing it compactly is:

(4.5) $\square\,(\forall\, p, q \in \mathit{Procs} \,:\, (p \neq q) \Rightarrow (\{pc(p), pc(q)\} \neq \{cs\})\,)$

We will not yet state a precise liveness condition a mutual exclusion algorithm should satisfy. All we need to know for now is that if some processes enter the waiting section, they can't all wait forever without entering the critical section.

Most people viewing the outline in Figure 4.1 will think this is an unrealistic description of a mutual exclusion algorithm because, by describing

```
**variables** $x = (p \in \{0, 1\} \mapsto \text{FALSE})$ ;
**process** $p \in \{0, 1\}$
   **variables** $pc = ncs$ ;
   **while** TRUE **do**
     $ncs$:  **skip** ;
     $wait$: $x[p] := \text{TRUE}$ ;
     $w2$:   **await** $\neg\, x[1 - p]$ ;
     $cs$:   **skip** ;
     $exit$: $x[p] := \text{FALSE}$
   **end while**
**end process**
```

Figure 4.2: The unacceptable mutual exclusion algorithm *UM*.

the execution of the critical section with a single **skip** step, we are assuming that the entire critical section is executed as a single step. Of course, we realize that this isn't the case. It no more says that the critical section is executed as a single step than our description of an hour-minute clock says that nothing else happens between the step that changes the clock's display to 7:29 and the step that changes it to 7:30. Just as 59 changes to a seconds display can occur between those two steps, process p can print the entire Bhagavad Gita while $pc(p)$ equals pc. A mutual exclusion algorithm simply describes all that printing as stuttering steps of the algorithm.

Figure 4.2 describes a program named *UM*, which is an abbreviation of *Unacceptable Mutual exclusion algorithm*. Technically, it's a mutual exclusion algorithm because it satisfies property (4.5) with *Procs* equal to the set $\{0, 1\}$ of processes. But for reasons that will be discussed later, it isn't considered to be an acceptable algorithm.

This pseudocode program is the first one we've seen with an **await** statement. For a state predicate P, the statement **await** P can be executed only when control is at the statement and P equals TRUE. We could write the statement a : **await** P as:

a: **if** $\neg P$ **then goto** a **end if**

Executing this statement in a state with P equal to TRUE just moves control to the next statement. Executing it in a state with P equal to FALSE does not change the value of any program variable, so it's a stuttering step of the program. Since a stuttering step is always allowed, executing the statement **await** P when P equals FALSE is the same as not executing it. So, while we can think of the statement **await** P continually evaluating the expression

P and moving to the next statement iff it finds P equal to TRUE, mathematically that's equivalent to describing it as an action A such that $\mathbb{E}(A)$ equals $(pc = a) \wedge P$.

This is also the first pseudocode we've seen with explicit array variables. An array variable x is an array-valued variable, where an array is a function and $x[p]$ just means $x(p)$. We've already seen implicit array variables—namely, the local variables t and pc of program *Increment* are represented by function-valued variables in Figure 3.5. I have decided to write $x[p]$ instead of $x(p)$ in pseudocode to make the pseudocode look more like real code. However, the value of an array variable can be any function, not just a finite ordinal sequence; and we write $x(p)$ instead of $x[p]$ when discussing the program mathematically. As we've seen in Figure 3.5, an assignment statement $x[p] :=\ \ldots$ is described mathematically as $x' = (x \text{ EXCEPT } p \mapsto \ldots)$.

Algorithm *UM* is quite simple. The processes communicate through the variable x, with process p writing to $x(p)$. The initial value of $x(p)$ for each process p is FALSE. To enter the critical section, process p sets $x(p)$ to TRUE and then enters its critical section when $x(1-p)$ (the array element written by the other process) equals FALSE.

It's easy to see that the two processes cannot be in their critical sections at the same time. If they were, the last process p to enter its critical section would have read $x(1-p)$ equal to TRUE when executing statement $w2$, so it couldn't have entered its critical section. Since mutual exclusion is an invariance property, it can be proved mathematically by finding an inductive invariant that implies mutual exclusion. You can check that the following formula is such an inductive invariant of *UM*:

$$(4.6)\quad \begin{array}{l} \wedge\ \mathit{TypeOK} \\ \wedge\ \forall\, p \in \{0,1\} : \begin{array}{l} \wedge\ (pc(p) \in \{w2, cs\}) \Rightarrow x(p) \\ \wedge\ (pc(p) = cs) \Rightarrow (pc(1-p) \neq cs) \end{array} \end{array}$$

where *TypeOK* is the type-correctness invariant:

$$\mathit{TypeOK} \triangleq \begin{array}{l} \wedge\ x \in (\{0,1\} \rightarrow \{\text{TRUE}, \text{FALSE}\}) \\ \wedge\ pc \in (\{0,1\} \rightarrow \{ncs, wait, w2, cs, exit\}) \end{array}$$

Let *UMSafe* be the safety property described by the pseudocode. We want to conjoin a property *UMLive* to *UMSafe* to state a fairness requirement of the program's behaviors. Let's make the obvious choice of defining *UMLive* to be formula (4.4) with *Procs* equal to $\{0,1\}$ and v equal to $\langle x, pc \rangle$. This implies that both processes keep taking steps forever, executing their critical sections infinitely often, which makes it seem like a good choice. Actually, that makes it a bad choice.

Algorithm *UM* is unacceptable because formula *UMSafe*, which describes the pseudocode, permits deadlock. If both processes execute statement *wait* before either executes $w2$, the algorithm reaches the deadlocked state in which neither **await** statement is enabled. Conjoining *UMLive* to *UMSafe* produces a formula asserting that such a deadlocked state cannot occur. It ensures the liveness property we want, that processes keep executing their critical sections. However, it does this not by requiring only that processes keep taking steps, but also by preventing them from taking some steps—namely, ones that produce a deadlocked state. A fairness property shouldn't do that.

Before going further, let's see why $\mathit{UMSafe} \wedge \mathit{UMLive}$ doesn't allow such a deadlocked state to be reached. The reason is that the formula satisfies this invariant:

(4.7) $\neg(pc(0) = pc(1) = w2)$

This is an invariant of $\mathit{UMSafe} \wedge \mathit{UMLive}$ because it is true initially and it can be made false only by a step in which a process p executes its *wait* statement in a state s with $pc(p-1) = w2$; and we now show that such a step cannot occur.

It's an invariant of *UMSafe*, and hence of $\mathit{UMSafe} \wedge \mathit{UMLive}$, that $pc(p) = \mathit{wait}$ implies $x(p) = \text{FALSE}$. Hence, $pc(p-1) = w2$ and $x(p) = \text{FALSE}$ in state s, which implies $\mathbb{E}\langle \mathit{PNext}(1-p)\rangle_v$ is true. Therefore, *UMLive* implies that $\diamond\langle \mathit{PNext}(1-p)\rangle_v$ must be true at state s of the behavior. This implies that the process p step can't occur, because it would lead to deadlock which would make such a $\langle \mathit{PNext}(1-p)\rangle_v$ step impossible. Therefore, no step of the program can make (4.7) false. Since it is true in an initial state, (4.7) is an invariant of $\mathit{UMSafe} \wedge \mathit{UMLive}$.

Thus, *UMLive* should not be the fairness property for algorithm *UM*, because it disallows a program step allowed by *UMSafe*. Before determining what the fairness property should be, let's characterize exactly what's wrong with property *UMLive*.

4.2.2.2 Machine Closure

The general principle illustrated by program *UM* is that fairness for a program should require only that something eventually happens, so it should rule out only infinite behaviors in which that thing never happens. It should not rule out finite behaviors.

A liveness property by itself does not rule out any finite behaviors. A liveness property L by definition allows any finite behavior to be completed to a behavior that satisfies L. Property *UMLive* is a liveness property, so the finite behavior in which the program reaches a deadlocked state can be completed to a behavior satisfying *UMLive*. For example, we can concatenate to that finite behavior a complete behavior satisfying $UMSafe \wedge UMLive$. (There are many behaviors of *UMSafe* that don't deadlock.) Since that concatenation contains infinitely many steps of each process, it satisfies *UMLive*. However, it doesn't satisfy *UMSafe* because the step between the last state of the deadlocked finite behavior (which is a deadlocked state) and the first state of a complete behavior does not satisfy the program's next-state action. It is impossible to complete that deadlocked behavior to a behavior satisfying both *UMLive* and *UMSafe* because the next-state action of *UMSafe* does not allow any non-stuttering step from a deadlocked state.

For a liveness property L to be a fairness property for *UMSafe*, it should not just require that any finite behavior can be completed to a behavior satisfying L; it should require that any finite behavior that satisfies *UMSafe* can be completed to a behavior that satisfies $UMSafe \wedge L$.

In general, a pair $\langle S, L\rangle$, where S is a safety property and L a liveness property, is defined to be *machine closed* iff every behavior satisfying S can be completed to a behavior satisfying $S \wedge L$. We require that a fairness property for a program having the safety property S be a liveness property L such that $\langle S, L\rangle$ is machine closed. The following theorem, proved in the Appendix, provides a nice mathematical characterization of machine closure.

Theorem 4.4 If S is a safety property and L a liveness property, then $\langle S, L\rangle$ is machine closed iff $\models \mathcal{C}(S \wedge L) \equiv S$.

4.2.3 Weak Fairness

Let's now see how to describe fairness with a machine-closed liveness property. Using the requirement

(4.8) $\mathbb{E}\langle PNext(p)\rangle_v \leadsto \langle PNext(p)\rangle_v$

worked fine for program *Increment*. It failed for program *UM*. The reason it worked for program *Increment* is that when $PNext(p)$ is enabled, it remains enabled until a $PNext(p)$ step occurs. In program *UM*, process p can reach a state in which $pc(p)$ equals $w2$ and $PNext(p)$ is enabled, but process $1-p$ can then take a step that disables action $PNext(p)$.

To obtain a machine-closed condition, we have to weaken (4.8) so it rules out fewer behaviors. The obvious way to do that is by requiring a $PNext(p)$ step to occur not if $PNext(p)$ just becomes enabled, but only if it remains enabled until a $PNext(p)$ step occurs. Let $F \,\mathcal{U}\, G$ be the temporal formula asserting that F is true until G is true. (We'll discuss later exactly what $\mathcal{U}$ means.) As our choice of fairness property, we will replace property (4.8) by:

(4.9) $(\, \mathbb{E}\langle PNext(p)\rangle_v \,\mathcal{U}\, \langle PNext(p)\rangle_v \,) \;\leadsto\; \langle PNext(p)\rangle_v$

However, rather surprisingly, (4.9) is equivalent to:

(4.10) $\Box\mathbb{E}\langle PNext(p)\rangle_v \;\leadsto\; \langle PNext(p)\rangle_v$

To prove this, we first have to examine the definition of $\mathcal{U}$. When we say that E is true until P is true, we usually mean that P is eventually true and E is true until P becomes true. But if $E \,\mathcal{U}\, P$ implies that $\Diamond P$ is true, then (4.9) would be trivially true and thus useless, since it would assert that $\langle PNext(p)\rangle_v$ conjoined with some other condition leads to $\langle PNext(p)\rangle_v$. So we have to interpret $E \,\mathcal{U}\, P$ to mean that either P is eventually true and E is true until it is, or P is not eventually true and E is true forever. Thus, whatever the precise meaning of $\mathcal{U}$ is, we have:

(4.11) $\models (E \,\mathcal{U}\, P) \;\equiv\; (\Diamond P \wedge E \,\mathcal{U}\, P) \vee (\neg\Diamond P \wedge \Box E)$

The equivalence of (4.9) and (4.10) follows from this RTLA theorem, which is proved in the Appendix.

Theorem 4.5 (4.11) implies $\models ((E \,\mathcal{U}\, P) \leadsto P) \;\equiv\; (\Box E \leadsto P)$ for any formulas E and P.

A liveness property commonly assumed of multiprocess algorithms, called *weak fairness*, is that (4.10) is true for every process p. We generalize this concept to define weak fairness of an arbitrary action A to be the formula $\mathrm{WF}_v(A)$ defined by:

(4.12) $\mathrm{WF}_v(A) \;\triangleq\; \Box\mathbb{E}\langle A\rangle_v \leadsto \langle A\rangle_v$

Another form of fairness called strong fairness that is sometimes assumed is discussed later. We will see how weak and strong fairness are used to write machine-closed descriptions of abstract programs. A special case of the general result is that if S is the formula $Init \wedge \Box[Next]_v$ and $Next$ equals $\exists\, p \in Procs : PNext(p)$, then $\langle S, \forall\, p \in Procs : \mathrm{WF}_v(PNext(p)\rangle$ is machine closed. But before we get to all that, let's examine weak fairness.

The first thing we observe is that for a multiprocess program described with pseudocode, weak fairness of a process's next-state action is equivalent to the conjunction of weak fairness of all the actions described by the process's statements. That is, if $\mathit{PNext}(p)$ equals $\exists\, i \in I : A_i(p)$ for a set I, action $A_i(p)$ describes the statement with label i, and v is the tuple of program variables, then:

$$(4.13) \quad \models \mathrm{WF}_v(\mathit{PNext}(p)) \;\equiv\; \forall\, i \in I \,:\, \mathrm{WF}_v(A_i(p))$$

This is because $\langle\, \mathit{PNext}(p) \,\rangle_v$ is enabled iff $\langle\, A_{pc(p)}(p) \,\rangle_v$ is enabled, in which case $pc(p)$ can be changed only by an $\langle\, A_{pc(p)}(p) \,\rangle_v$ step. Thus, in any state of a behavior with $pc(p) = i$, the formula $\mathbb{E}\langle\, \mathit{PNext}(p) \,\rangle_v \Rightarrow \Diamond\langle\, \mathit{PNext}(p) \,\rangle_v$ is equivalent to $\mathbb{E}\langle\, A_i(p) \,\rangle_v \Rightarrow \Diamond\langle\, A_i(p) \,\rangle_v$. A rigorous justification of (4.13) is that it is a special case of Theorem 4.7 in Section 4.2.7 below.

The following tautology is useful for deducing properties from weak fairness assumptions:

$$(4.14) \quad \models \mathrm{WF}_v(A) \;\equiv\; (\Diamond\Box\,\mathbb{E}\langle\, A \,\rangle_v \Rightarrow \Box\Diamond\langle\, A \,\rangle_v)$$

It makes weak fairness look stronger than the definition because $\Box\Diamond\langle\, A \,\rangle_v$ is a stronger property than $\Diamond\langle\, A \,\rangle_v$. Here's an informal proof of (4.14). The definition of $\mathrm{WF}_v(A)$ implies the right-hand side of the equivalence because $\Diamond\Box\,\mathbb{E}\langle\, A \,\rangle_v$ implies that eventually $\langle\, A \,\rangle_v$ is always enabled, whereupon $\mathrm{WF}_v(A)$ keeps forever implying that an $\langle\, A \,\rangle_v$ step occurs, so there must be infinitely many $\langle\, A \,\rangle_v$ steps, making $\Box\Diamond\langle\, A \,\rangle_v$ true. The opposite implication is true because $\Box\,\mathbb{E}\langle\, A \,\rangle_v$ implies $\Diamond\Box\,\mathbb{E}\langle\, A \,\rangle_v$, so the right-hand side of the equivalence implies that $\Box\Diamond\langle\, A \,\rangle_v$ is true and hence $\Diamond\langle\, A \,\rangle_v$ is true. A rigorous proof of (4.14) is by the following RTLA[4] reasoning, substituting $\mathbb{E}\langle\, A \,\rangle_v$ for F and $\langle\, A \,\rangle_v$ for G:

$$\begin{array}{rcll}
\Box F \leadsto G & \equiv & \Box(\Box F \Rightarrow \Diamond G) & \text{by definition of } \leadsto \\
& \equiv & \Box(\neg\Box F \lor \Diamond G) & \text{by propositional logic} \\
& \equiv & \Box(\Diamond\neg F \lor \Diamond G) & \text{by } \neg\Box F = \Diamond\neg F \text{ (3.25)} \\
& \equiv & \Box\Diamond(\neg F \lor G) & \text{by (3.23)} \\
& \equiv & \Box\Diamond\neg F \lor \Box\Diamond G & \text{by (3.27)} \\
& \equiv & \neg\Box\Diamond\neg F \Rightarrow \Box\Diamond G & \text{by propositional logic} \\
& \equiv & \Diamond\Box F \Rightarrow \Box\Diamond G & \neg\Box\Diamond\neg F \equiv \Diamond\neg\Diamond\neg F \equiv \Diamond\Box\neg\neg F \text{ by (3.25)}
\end{array}$$

The following tautology is useful for proving a weak-fairness formula, because it has an additional hypothesis in the implication:

$$(4.15) \quad \models \mathrm{WF}_v(A) \;\equiv\; (\Box\,\mathbb{E}\langle\, A \,\rangle_v \land \Box[\neg A]_v \leadsto \langle\, A \,\rangle_v)$$

[4]With these definitions of F and G, the formula $\Box\Diamond(\neg F \lor G)$ is not a TLA formula.

It is proved by expanding the definition of $\leadsto$ and applying the propositional logic tautology $\models (F \Rightarrow G) \equiv (F \wedge \neg G \Rightarrow G)$ and the TLA tautology $\models \neg\Diamond\langle A\rangle_v \equiv \Box[\neg A]_v$. Using (4.15) to prove $\mathrm{WF}_v(A)$ is essentially a proof by contradiction.

Proving these kinds of temporal logic tautologies is a good exercise. However, there are temporal logic theorem provers that can do it for you.

4.2.4 Temporal Logic Reasoning

Thus far, the only properties we've verified that programs satisfy have been invariance properties. Proving invariance requires no temporal logic reasoning. To prove $\models \mathit{Init} \wedge \Box[\mathit{Next}]_v \Rightarrow \Box\mathit{Inv}$ we prove the LA formulas $\models \mathit{Int} \Rightarrow \mathit{Inv}$ and $\models \mathit{Inv} \wedge [\mathit{Next}]_v \Rightarrow \mathit{Inv}'$ and then apply a single temporal logic proof rule.

Nontrivial temporal logic reasoning is required for proving that programs satisfy liveness properties. We often prove liveness properties of the form $P \leadsto Q$. This property asserts that something is true at all "times" in a behavior—namely, whenever P is true, Q is eventually true. The description $\mathit{Init} \wedge \Box[\mathit{Next}]_v$ of a program cannot be used directly to prove $P \leadsto Q$ because it asserts only that something is true initially. For that reason, the first thing we do when proving $P \leadsto Q$ is to prove that some formula Inv is an invariant of the program, so the program implies $\Box\mathit{Inv} \wedge \Box[\mathit{Next}]_v$. We then use $\Box\mathit{Inv} \wedge \Box[\mathit{Next}]_v$ to prove $P \leadsto Q$.

A formula F that asserts something is true at all times is called a $\Box$ formula. The formula $\Box\mathit{Inv} \wedge \Box[\mathit{Next}]_v$ is a $\Box$ formula because it's equivalent to the RTLA formula $\Box(\mathit{Inv} \wedge [\mathit{Next}]_v)$. Because of the tautology $\models \Box\Box F \equiv \Box F$, we can define F to be a $\Box$ formula iff it's equivalent to $\Box F$. By (3.17), the conjunction of $\Box$ formulas is a $\Box$ formula. In general, (3.18) implies $\forall\, i \in S : F_i$ is a $\Box$ formula if each F_i is a $\Box$ formula.

In a proof, we almost always want every temporal logic formula asserted by a statement to be a $\Box$ formula. A theorem with statement F asserts $\models F$. The proof rule (3.16) tells us that $\models F$ implies $\models \Box F$. This means that whenever we prove F, we have proved $\Box F$. However, that does not mean that when we have proved a step in a proof that asserts F, we have proved $\Box F$. When we prove a step

4.2.7. $G \leadsto H$

we have not proved $\models G \leadsto H$. We have proved $\models \mathit{Asp} \Rightarrow (G \leadsto H)$, where Asp is the conjunction of all the assumptions in effect at statement 4.2.7. By the proof rule (3.21), $\models \mathit{Asp} \Rightarrow (G \leadsto H)$ implies $\models \Box\mathit{Asp} \Rightarrow \Box(G \leadsto H)$.

If Asp is a $\Box$ formula, so it's equivalent to $\Box Asp$, then proving 4.2.7 proves $\models Asp \Rightarrow \Box(G \leadsto H)$. Therefore proving 4.2.7 is equivalent to proving

4.2.7. $\Box(G \leadsto H)$

if Asp is a $\Box$ formula. Since the conjunction of $\Box$ formulas is a $\Box$ formula, Asp is a $\Box$ formula if all the assumptions in effect at statement 4.2.7 are $\Box$ formulas.

We assure that every statement that asserts a temporal formula F asserts $\Box F$ by making every temporal formula in an ASSUME clause be a $\Box$ formula. Any temporal formula F asserted by a statement can then be considered to assert $\Box F$. The following are $\Box$ formulas for all temporal formulas F and G, state expressions v, and actions A: $\Box\Diamond F$ (obviously), $\Diamond\Box F$ by (3.29), $F \leadsto G$ by definition of $\leadsto$, and $\mathrm{WF}_v(A)$ by the definition (4.12) of WF.

4.2.5 Reasoning With Weak Fairness

We now see how to show that an abstract program with weak fairness conditions satisfies a liveness property. We will do this with a modification of algorithm *UM* called the *One-Bit* algorithm that is an acceptable mutual exclusion algorithm. But first, we examine what liveness condition the algorithm should satisfy.

4.2.5.1 Liveness for Mutual Exclusion

The liveness condition Dijkstra required of the mutual exclusion algorithm outlined in Figure 4.1 was that if some process is at statement *wait*, then eventually some process enters its critical section—expressed by:

(4.16) $(\exists\, p \in Procs : pc(p) = wait) \leadsto (\exists\, p \in Procs : pc(p) = cs)$

This condition is usually called *deadlock freedom*. That's a misuse of the term, because deadlock freedom is actually the safety property asserting that the program never reaches a deadlocked state—one in which no process can take a step. Property (4.16) also rules out what is called *livelock*, in which no process enters its critical section although some processes keep executing statements in their waiting sections. However, when discussing mutual exclusion, we will use deadlock freedom to mean property (4.16). This condition allows one or more processes to be starved—that is, to remain forever in their waiting section—while other processes enter and leave the critical section.

```
variables $x = (p \in \{0,1\} \mapsto$ FALSE$)$ ;
process $p \in \{0,1\}$
  variables $pc = ncs$ ;
  while TRUE do
    $ncs$:  skip ;
    $wait$: $x[p]$ := TRUE ;
    $w2$:   if $p = 0$ then await $\neg x[1]$
                    else  if $x[0]$ then $w3$: $x[1]$ := FALSE ;
                                         $w4$: await $\neg x[0]$ ;
                                               goto $wait$
                          end if
            end if ;
    $cs$:   skip ;
    $exit$: $x[p]$ := FALSE
  end while
end process
```

Figure 4.3: Algorithm *OB*.

Dijkstra also required that processes be allowed to remain forever in their noncritical sections. Just because a process might send output to the printer, we don't want to insist that it does. This requirement rules out simple algorithms in which processes take turns entering the critical section. A process that does not want to enter its critical section cannot be required to do anything to allow other processes to enter their critical sections.

4.2.5.2 The One-Bit Algorithm

The basic idea of the One-Bit algorithm is to modify algorithm *UM* to prevent deadlock by having process 1 wait when both processes are concurrently trying to enter the critical section. This is done by modifying process 1 so that if it sees that $x[0]$ equals TRUE in statement $w2$, then it sets $x[1]$ to FALSE and waits until $x[0]$ equals FALSE (so process 0 has exited its critical section) before going back to statement *wait* and trying again. The pseudocode for the algorithm, which we call *OB* is in Figure 4.3.[5]

Algorithm *OB* satisfies mutual exclusion because processes use the same protocol to enter the critical section as algorithm *UM*: Each process p sets $x[p]$ to TRUE and can then enter the critical section only if $x[1-p]$ equals

[5]Algorithm *OB* is the two-process case of an N-process algorithm that was discovered independently by James E. Burns and me in the 1970s, but not published until later [7, 29].

FALSE. In fact, it has the same inductive invariant (4.6) as *UM* except that the type invariant *TypeOK* must be modified because $pc(1)$ can now also equal $w3$ or $w4$. For algorithm *OB*, we define:[6]

$$(4.17)\quad \mathit{TypeOK} \;\triangleq\; \begin{array}{l} \land\; x \in (\{0,1\} \to \{\text{TRUE}, \text{FALSE}\}) \\ \land\; pc \in (\{0,1\} \to \{ncs, wait, w2, w3, w4, cs, exit\}) \\ \land\; pc(0) \notin \{w3, w4\} \end{array}$$

We define *Inv* to equal (4.6), with *TypeOK* defined by (4.17).

Let *OBSafe*, the safety property of *OB* described by the pseudocode, be the formula $\mathit{Init} \land \Box[\mathit{Next}]_v$, where v equals $\langle x, pc\rangle$ and

$$\mathit{Next} \;\triangleq\; \exists\, p \in \{0,1\} \,:\, \mathit{PNext}(p)$$

The fairness condition we want *OB* to satisfy is weak fairness of each process's next-state action except when the process is in its noncritical section. A process p remaining forever in its noncritical section is represented in our abstract program by no $\mathit{PNext}(p)$ step occurring when $pc(p)$ equals ncs. The fairness condition we assume of program *OB* is therefore:

$$\mathit{OBFair} \;\triangleq\; \forall\, p \in \{0,1\} \,:\, \mathrm{WF}_v((pc(p) \neq ncs) \land \mathit{PNext}(p))$$

The formula $\mathit{OBSafe} \land \mathit{OBFair}$, which we call *OB*, satisfies the liveness property that if process 0 is in its waiting section, then it will eventually enter its critical section. That is, *OB* implies:

$$(4.18)\quad (pc(0) \in \{wait, w2\}) \leadsto (pc(0) = cs)$$

This implies deadlock freedom, because if process 0 stops entering and leaving its critical section, then it eventually stays forever in its noncritical section. If process 1 is then in its waiting section, it will read $x[0]$ equal to FALSE and enter its critical section.

4.2.5.3 Proving Liveness

We will now see how to reason more rigorously about liveness. Even if you never write rigorous correctness proofs, learning how to reason about liveness will help you better understand liveness properties.

There are two kinds of liveness properties that we prove: that a program implies leads-to properties such as (4.16), and that a program implies the

[6] For any infix predicate symbol like $=$ or $\in$, putting a slash through the symbol negates it, so $e \notin S$ means $\neg(e \in S)$.

$\Box \mathit{Inv} \land \Box[\mathit{Next}]_v \land \mathit{OBFair}$

$$\begin{tikzcd}
pc(0) \in \{wait, w2\} \arrow[rr, "1"] \arrow[dr, "1"'] & & pc(0) = w2 \arrow[r, "3"] & pc(0) = cs \\
& pc(0) = wait \arrow[ur, "2"'] & &
\end{tikzcd}$$

Figure 4.4: Leads-to lattice for the proof of (4.18).

fairness properties of a more abstract program. Here we consider leads-to properties. Proving fairness properties is discussed in Section 5.4.

The proof of a leads-to formula is usually decomposed into proving simpler leads-to formulas. Figure 4.4 shows how we decompose the proof of formula (4.18) using what we call a *leads-to lattice*.

First, let's pretend that the box and the formula labeling it aren't there. We then have just a directed graph whose nodes are formulas. A formula F and its outgoing edges represent the assertion that F leads to the disjunction of the formulas those edges point to. Thus, the two edges numbered 1 assert the formula:

$$(pc(0) \in \{wait, w2\}) \;\leadsto\; ((pc(0) = wait) \lor (pc(0) = w2))$$

By the meaning of *leads to*, the property asserted by each formula F in the graph means that if the program is ever in a state for which F is true, then it will eventually be in a state satisfying a formula pointed to by one of the outgoing edges from F. The graph has a single sink node (one having no outgoing edge). Every path in the graph, if continued far enough, leads to the sink node. By transitivity of the $\leadsto$ relation, this means that if all the properties asserted by the diagram are true of a behavior, then the behavior satisfies the property $F \leadsto H$, where H is the sink-node formula and F is any formula in the lattice. In particular, the properties asserted by the diagram imply formula (4.18). By (3.31), that every formula in the graph leads to the sink-node formula means that the disjunction of all the formulas in the graph leads to the sink-node formula.

Now to explain the box. Let Λ equal $\Box \mathit{Inv} \land \Box[\mathit{Next}]_v \land \mathit{OBFair}$, the formula that labels the box. Formula Λ is implicitly conjoined to each of the formulas in the graph. It is a $\Box$ formula, since the conjunction of $\Box$ formulas is a $\Box$ formula, and *OBFair* is the conjunction of WF formulas, which are $\Box$ formulas.

Since Λ is conjoined to every formula in it, the leads-to lattice makes assertions of the form

$$\Lambda \wedge G \;\leadsto\; (\Lambda \wedge H_1) \vee \ldots \vee (\Lambda \wedge H_j)$$

Since Λ equals $\Box\Lambda$, and once $\Box\Lambda$ is true it is true forever, this formula is equivalent to $\Lambda \wedge G \;\leadsto\; H_1 \vee \ldots \vee H_j$. (This follows from (3.32c).and propositional logic.)

If H is the unique sink node of the lattice, then proving the assertions made by the lattice proves $\models \Lambda \wedge G \leadsto H$ for every node G of the lattice. Since Λ equals $\Box\Lambda$, it's easy to see that a behavior that satisfies $\Lambda \wedge G \leadsto H$ must satisfy $\Lambda \Rightarrow (G \leadsto H)$, so proving $\models \Lambda \wedge G \leadsto H$ proves $\models \Lambda \Rightarrow (G \leadsto H)$. All this is true only because the formula Λ is a $\Box$ formula. In general, we label a box in a leads-to lattice only with a $\Box$ formula.

Remember what the proof lattice of Figure 4.4 is for. We want to prove that *OB* implies (4.18). By proving the assertions made by the proof lattice, we show that the formula Λ labeling the box implies (4.18). By definition of *OB* and because *OB* implies $\Box Inv$, formula Λ is implied by *OB*. Therefore, by proving the leads-to properties asserted by the proof lattice, we prove that *OB* implies (4.18). Note how we had to use the $\Box$ formula $\Box Inv \wedge \Box[Next]_v$ instead of *OBSafe*, which is true only initially.

To complete the proof that *OB* implies (4.18), we now prove the leads-to properties asserted by Figure 4.4. The leads-to property asserted by the edges numbered 1 is:

$$\Lambda \,\wedge\, (pc(0) \in \{wait, w2\}) \;\leadsto\; ((pc(0) = wait) \vee (pc(0) = w2))$$

It is trivially true, since $pc(0) \in \{wait, w2\}$ implies that $pc(0)$ equals *wait* or $w2$, and $\Box(F \Rightarrow G)$ implies $F \leadsto G$.

The formula $\Lambda \wedge (pc(0) = wait) \leadsto (pc(0) = w2)$ asserted by edge number 2 is true because Λ implies $\Box Inv \wedge \Box[Next]_v$ and the weak fairness assumption of process 0, which imply $(pc(0) = wait) \leadsto (pc(0) = w2)$.

The formula

$$(4.19)\quad \Lambda \wedge (pc(0) = w2) \;\leadsto\; (pc(0) = cs)$$

asserted by edge number 3 is the interesting one. Its proof is decomposed with the proof lattice of Figure 4.5.

The property asserted by the edges numbered 1 in this leads-to lattice has the form $\Lambda \wedge F \leadsto (G \vee (F \wedge \Box\neg G))$. This formula is a tautology. Intuitively, it's true because if F is true now, then either G is true eventually or F is true now and $\neg G$ is true from now on. It's proved by:

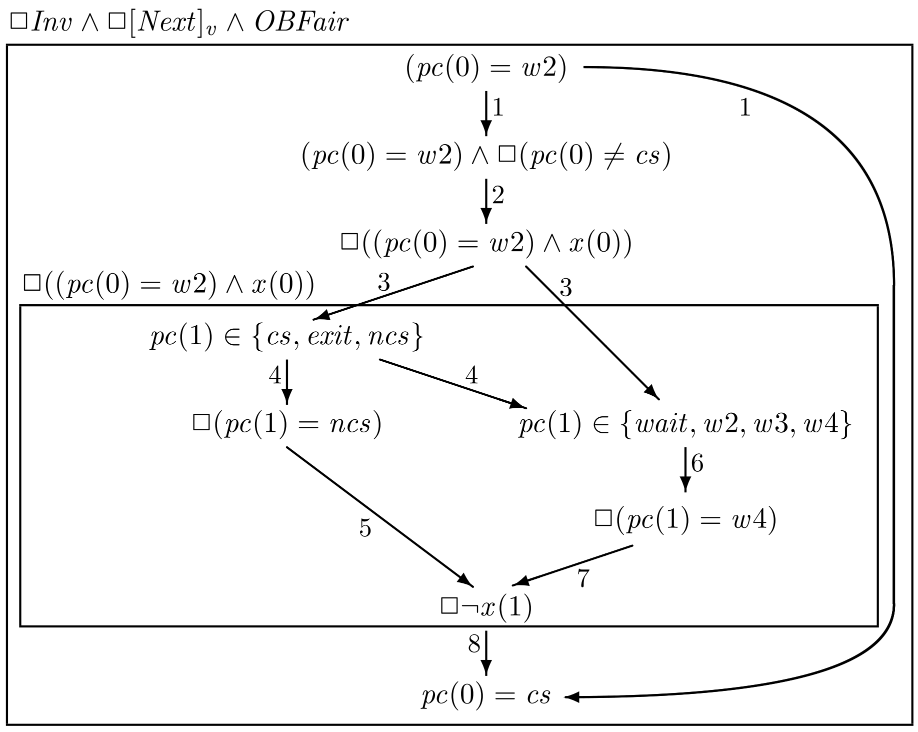
$\Box Inv \land \Box[Next]_v \land OBFair$
$(pc(0) = w2)$
1
1
$(pc(0) = w2) \land \Box(pc(0) \neq cs)$
2
$\Box((pc(0) = w2) \land x(0))$
$\Box((pc(0) = w2) \land x(0))$
3
3
$pc(1) \in \{cs, exit, ncs\}$
4
4
$\Box(pc(1) = ncs)$
$pc(1) \in \{wait, w2, w3, w4\}$
6
$\Box(pc(1) = w4)$
5
7
$\Box\neg x(1)$
8
$pc(0) = cs$

Figure 4.5: Leads-to lattice for the proof of (4.19).

$$
\begin{array}{rcll}
\Lambda \land F & \Rightarrow & F \land (\Diamond G \lor \Box\neg G) & \Diamond G \lor \Box\neg G \text{ equals TRUE} \\
& \Rightarrow & \Diamond G \lor (F \land \Box\neg G) & \text{by propositional logic} \\
& \Rightarrow & \Diamond G \lor \Diamond(F \land \Box\neg G) & \text{by (3.23)} \\
& \equiv & \Diamond(G \lor (F \land \Box\neg G)) & \text{by (3.23)}
\end{array}
$$

which shows $\models (\Lambda \land F \Rightarrow \Diamond(G \lor (F \land \Box\neg G))$, from which we can deduce $\models \Lambda \land F \leadsto (G \lor (F \land \Box\neg G))$ by (3.20) and the definition of $\leadsto$.

The leads-to formula asserted by edge 2 is an implication. It's true because $\Box Inv \land \Box[Next]_v$ implies that, if $pc(0) = w2$ and $\Box(pc(0) \neq cs)$ are ever true, then $pc(0) = w2$ must remain true forever, which by $\Box Inv$ implies $x(0)$ must equal TRUE forever. It is proved by proving that $(pc(0) = w2) \land x(0)$ is an invariant of an abstract program. The initial predicate of this program is $Inv \land (pc(0) = w2)$. Its next-state relation is:

$$Next2 \triangleq Inv \land Next \land (pc(0) \neq cs)'$$

The formula $\Box[Next2]_v$ is implied by $\Box Inv$ and $\Box[Next]_v$ and the conjunct $\Box(pc(0) \neq cs)$ of the formula at the tail of the edge 2 arrow. (Note that the prime in this formula is valid because $pc(0) \neq cs$ always true implies that it's always true in the next state.) We are using an invariance property of one program to prove a liveness property of another program. This would seem strange if we were thinking in terms of code. But we're thinking mathematically, and a mathematical proof contains lots of formulas. It's not surprising if one of those formulas looks like the formula that describes a program.

The edges numbered 3 enter a box whose label is the same formula from which those edges come. In general, an edge can enter a box with a label $\Box F$ if it comes from a formula that implies $\Box F$. This is because a box labeled $\Box F$ is equivalent to conjoining $\Box F$ to all the formulas in the box, and $\Box F \leadsto (G_1 \lor \ldots \lor G_n)$ implies $\Box F \leadsto ((\Box F \land G_1) \lor \ldots \lor (\Box F \land G_n))$. An arrow can always leave a box, since removing the formula it points to from the box just weakens that formula.

Proofs of the assertions represented by the rest of the lattice's edges are sketched below.

edges 3 The formula represented by these edges is true because the disjunction of the formulas they point to asserts that $pc(1)$ is in the set $\{ncs, wait, w2, w3, w4, cs, exit\}$, which is implied by $\Box Inv$.

edges 4 If $pc(1)$ equals cs or $exit$, then $\Box Inv \land \Box[Next]_v$ and the fairness condition for process 1 imply that it will eventually be at ncs. Either $pc(1)$ equals ncs forever or eventually it will not equal ncs. In the latter case, $\Box[Next]_v$ implies that the step that makes $pc(1) = ncs$ false must make $pc(1) = wait$ true.

edge 5 This is an implication since $\Box Inv$ implies that if process 1 is forever at ncs, then $x(1)$ is forever false.

edge 6 If process 1 is at $wait$, $w2$, or $w3$, then its weak fairness condition implies it is eventually at $w4$. When process 1 is at $w4$, formulas $\Box[Next]_v$ and $\Box x(0)$ (from the label of the inner box) imply that it must remain forever at $w4$.

edge 7 This is an implication, because Inv and $pc(1) = w4$ imply $\neg x(1)$.

edge 8 $\Box\neg x(1)$, $\Box(pc(0) = w2)$ (implied by the inner box's label), and $OBFair$ imply that a process 0 step that makes $pc(0)$ equal to cs must eventually occur. (Equivalently, these three formulas are contradictory, so they imply FALSE which implies anything.)

The proof sketches of the properties asserted by edges 4 and edge 6 skim over more details than the proofs of the other the properties asserted by the lattice. A more detailed proof would be described by a lattice in which each of the formulas pointed to by the edges numbered 3 were split into multiple formulas—for example, the formula $pc(1) \in \{cs, exit, ncs\}$ would be split into the formulas $pc(1) = cs$, $pc(1) = exit$, and $pc(1) = ncs$. A good check of your understanding is to draw the more detailed lattice and write proof sketches for its new edges.

4.2.6 Strong Fairness

4.2.6.1 Starvation Free Mutual Exclusion

Mutual exclusion was not motivated by sharing a printer. It's needed when multiple processes perform operations on the same data. As we saw from the *Increment* example of Section 3.3, even sharing a simple counter without synchronization can result in increment operations being lost. An easy way to synchronize data sharing is to put every operation to the shared data in a critical section.

The One-Bit Algorithm *OB* implements mutual exclusion with processes that communicate using only simple reads and writes of shared variables. Synchronizing processes in this way is inefficient. Dijkstra proposed the communication mechanism called a *binary semaphore* or *lock*. A lock is a variable that can have two values, traditionally taken to be 0 and 1. Let's call that variable *sem*. Initially *sem* equals 1. A process can execute two atomic operations, $P(sem)$ and $V(sem)$, to the lock. These operations can be described in pseudocode as:

```
$\underline{P(sem)}$                 $\underline{V(sem)}$
**await** $sem = 1$;       $sem := 1$
$sem := 0$
```

Locks were originally implemented with operating system calls. Modern multiprocessor computers provide machine instructions to implement them. Using a lock, mutual exclusion for any set *Procs* of processes can be implemented with the trivial algorithm *LM* of Figure 4.6.

Let $PNext(p)$ now be the next-state action of process p of program LM. With weak fairness of $(pc(p) \neq ncs) \wedge PNext(p)$ for each process p as its fairness property, algorithm LM satisfies the deadlock freedom condition (4.16). However, deadlock freedom allows individual processes to be starved, remaining forever in the waiting section.

```
**variables** $sem = 1$ ;
**process** $p \in Procs$
  **while** TRUE **do**
    $ncs$:  **skip** ;
    $wait$: $P(sem)$ ;
    $cs$:   **skip** ;
    $exit$: $V(sem)$
  **end while**
**end process**
```

Figure 4.6: Program *LM*: mutual exclusion with a semaphore.

Let $Wait(p)$, $Cs(p)$, and $Exit(p)$ be the actions described by the statements in process p with the corresponding labels *wait*, *cs*, and *exit*. Weak fairness of $(pc(p) \neq ncs) \wedge PNext(p)$ is equivalent to the conjunction of weak fairness of the actions $Wait(p)$, $Cs(p)$, and $Exit(p)$. Program *LM* allows starvation of individual processes because weak fairness of the actions of each process ensures that if multiple processes are waiting to execute that action, then some process will execute it. But if processes continually reach the *wait* statement, some individual processes p may never get to execute $Wait(p)$.

It's reasonable to require the stronger condition of *starvation freedom*, which asserts that no process starves. This is the property

(4.20) $\forall\, p \in Procs \,:\, (pc(p) = wait) \leadsto (pc(p) = cs)$

which asserts that any process reaching *wait* must eventually enter its critical section. For *LM* to satisfy this property, it needs a stronger fairness property than weak fairness of the $Wait(p)$ actions.

4.2.6.2 The Definition of Strong Fairness

By the usual meaning of *fair*, the fairest lock would be one in which processes execute their $Wait(p)$ actions in the order in which they set $pc(p)$ to *wait*. Some implementations of locks ensure this property. However, we don't consider it to be a fairness property because it produces a description of program *LM* that is not machine closed. It rules out finite behaviors in which processes execute their $Wait(p)$ actions in the wrong order—executions in which a process p reaches the *wait* statement after process q, but p enters the critical section before q does. Machine closure means that the liveness condition does not forbid any finite behaviors allowed by the program's safety property described by the pseudocode.

There is a standard way to strengthen weak fairness that produces machine closed program descriptions. Weak fairness of an action A asserts that if ever A is always enabled, then an A action must eventually occur. To strengthen this condition, we replace the requirement that A be always enabled by the weaker requirement that it be infinitely often enabled. We therefore define *strong fairness* of A by:

(4.21) $\mathrm{SF}_v(A) \;\triangleq\; \Box\Diamond\mathbb{E}\langle A\rangle_v \leadsto \langle A\rangle_v$

Analogous to formulas (4.14) and (4.15) for weak fairness are:

(4.22) $\models \mathrm{SF}_v(A) \;\equiv\; (\Box\Diamond\mathbb{E}\langle A\rangle_v \Rightarrow \Box\Diamond\langle A\rangle_v)$

(4.23) $\models \mathrm{SF}_v(A) \;\equiv\; (\Box\Diamond\mathbb{E}\langle A\rangle_v \wedge \Box[\neg A]_v \leadsto \langle A\rangle_v)$

The informal justification and the proof of (4.22) are similar to the ones for (4.14). The proof of (4.23) is essentially the same as that of (4.15).

4.2.6.3 Using a Strongly Fair Semaphore

To make program *LM* starvation free, meaning that it satisfies (4.20), we conjoin to the safety property *LMSafe* defined by the pseudocode the fairness property *LMFair* equal to $\forall\, p \in \mathit{Procs} : \mathit{LMPFair}(p)$, where $\mathit{LMPFair}(p)$ is the fairness requirement for process p. If we let $\mathit{LMPFair}(p)$ be the conjunction of strong fairness of $\mathit{Wait}(p)$ and weak fairness of $\mathit{Cs}(p)$ and $\mathit{Exit}(p)$, then the following argument shows $\mathit{LMSafe} \wedge \mathit{LMFair}$ implies starvation freedom. Starvation means that a process p waits forever with $pc(p) = \mathit{wait}$. That is possible only if other processes keep entering and leaving their critical sections. But whenever a process executes the *Exit* action, it sets *sem* to 1, which makes $\mathbb{E}\langle\, \mathit{Wait}(p)\rangle_v$ true. Thus $\mathbb{E}\langle\, \mathit{Wait}(p)\rangle_v$ must be true infinitely often, which by strong fairness of $\mathit{Wait}(p)$ implies that $\mathit{Wait}(p)$ is eventually executed, so p must enter its critical section.

There are several ways of writing formula $\mathit{LMPFair}(p)$. First, we observe that weak and strong fairness are equivalent for the actions $\mathit{Cs}(p)$ and $\mathit{Exit}(p)$. This is because the action is enabled iff $pc(p)$ has the appropriate value, so it remains enabled until a step of that action occurs to change $pc(p)$. Thus, when the action is enabled, it is continuously enabled until it is executed. We can therefore write *LMFair* as the conjunction of strong fairness of the three actions $\mathit{Wait}(p)$, $\mathit{Cs}(p)$, and $\mathit{Exit}(p)$.

The same sort of reasoning that led to (4.13) of Section 4.2.3, as well as Theorem 4.7 of Section 4.2.7, imply that the conjunction of strong fairness of these three actions is equivalent to strong fairness of their disjunction. Therefore, we can write $LMFair$ as strong fairness of their disjunction, which equals $(pc(p) \neq ncs) \land PNext(p)$.

While $\mathrm{SF}_v((pc(p) \neq ncs) \land PNext(p))$ is compact, I prefer not to define $LMPFair(p)$ this way because it suggests to a reader of the formula that strong fairness of $Cs(p)$ and $Exit(p)$ is required, although only weak fairness is. Usually, the process's next-state action will be the disjunction of many actions, and strong fairness is required of only a few of them. I would define $LMFair$ to equal

$$\mathrm{WF}_v((pc(p) \neq ncs) \land PNext(p)) \ \land\ \mathrm{SF}_v(Wait(p))$$

This is redundant because the first conjunct implies weak fairness of $Wait(p)$ and the second conjunct asserts strong fairness of it. But a little redundancy doesn't hurt, and its redundancy should be obvious because strong fairness implies weak fairness.

4.2.7 Properties of WF and SF

Several assertions were made above about weak and strong fairness. We now assert the general theorems from which those assertions follow. The first is that taking weak and strong fairness conditions as the liveness property produces a machine-closed description of a program. This is true if we take not just a single weak or strong fairness property or the conjunction of a finite number of them. It is true for a conjunction of a countable set of weak and/or strong fairness conditions, if each of those conditions asserts fairness of a subaction of the program's next-state action.

An action A is defined to be a *subaction* of an action $Next$ iff $\models A \Rightarrow Next$ is true. The subactions of the next-state action $Next$ for which fairness is asserted are usually disjuncts in the definition of $Next$—for example, $Wait(p)$ is a subaction of the next-state action $Next$ of algorithm OB. Often, we assert fairness of an action $P \land A$ where A is a disjunct in the definition of $Next$ and P is a state predicate—for example, the action $(pc(p) \neq ncs) \land PNext(p)$ of OB.[7]

We now state the theorem asserting that the conjunction of fairness properties produces a machine-closed specification. Its proof is in the Appendix.

enlargethispage and newpage (to make link to theorem 4.6 work).

[7]All the results that will be stated are satisfied if we allow a subaction A to be one satisfying $\models Inv \land A \Rightarrow Next$, where Inv is an invariant of the program. However, this generalization does not seem to be needed in practice.

Theorem 4.6 Let *Init* be a state predicate, *Next* an action, and v a tuple of all variables occurring in *Init* and *Next*. If A_i is a subaction of *Next* for all i in a countable set I, then the pair

$$\langle\, Init \land \Box[Next]_v \,,\ \forall\, i \in I \,:\, \mathrm{XF}^i_v(A_i) \,\rangle$$

is machine closed, where each XF^i_v may be either WF_v or SF_v.

Writing an infinite conjunction of fairness properties may not seem to be something we would do in practice. However, the next-state action of an abstract program sometimes does contain an infinite disjunction—that is, existential quantification over an infinite set of subactions—and we might want a fairness condition for each of those subactions. For example, a program that dynamically creates processes may be described as having an infinite number of processes, only a finite number of which have their next-state action enabled at any time. We might want a fairness condition for each of those processes.

It was stated above that, for program LM, weak or strong fairness of $(pc(p) \neq ncs) \land PNext(p)$ is equivalent to weak or strong fairness of its three subactions $Wait(p)$, $Cs(p)$, and $Exit(p)$. That's true because when any one of these subactions is enabled, a step of neither of the other two subactions can occur until a step of that subaction occurs. Let Q equal $(pc(p) \neq ncs) \land PNext(p)$ and let's call its three subactions A_1, A_2, and A_3. This condition can then be asserted as:

$$(4.24)\quad \forall\, i \in 1\,..\,3 \,:\, \mathbb{E}\langle A_i\rangle_v \;\Rightarrow\; (\mathbb{E}\langle Q\rangle_v \Rightarrow \mathbb{E}\langle A_i\rangle)\,\mathcal{U}\,\langle A_i\rangle_v$$

where $\mathcal{U}$ is the *until* operator with which we first defined weak fairness of $PNext(p)$ as (4.8). Similarly to what we did for weak fairness, we can remove the $\mathcal{U}$ by observing that $F\ \mathcal{U}\ G$ implies that if G is never true, then F must remain true forever. That $\langle A_i\rangle_v$ is never true is asserted by $\neg\Diamond\langle A_i\rangle_v$, which is equivalent to $\Box[\neg A_i]_v$. Therefore (4.24) implies

$$(4.25)\quad \forall\, i \in 1\,..\,3 \,:\, \mathbb{E}\langle A_i\rangle_v \land \Box[\neg A_i]_v \;\Rightarrow\; \Box(\mathbb{E}\langle Q\rangle_v \Rightarrow \mathbb{E}\langle A_i\rangle)$$

While (4.24) implies (4.25), the formulas are not equivalent. Formula (4.25) is strictly weaker than (4.24). However, it's strong enough to imply that strong or weak fairness of all the A_i is equivalent to strong or weak fairness of Q—assuming that Q is the disjunction of the A_i. Here is the precise theorem; its proof is in the Appendix.

newpage to
make link to
theorem work

Theorem 4.7 Let A_i be an action for each $i \in I$, let $Q \triangleq \exists\, i \in I : A_i$, and let XF be either WF or SF. Then

$$\begin{aligned} \models\ & (\forall\, i \in I : \Box(\mathbb{E}\langle A_i\rangle_v \land \Box[\neg A_i]_v \Rightarrow \\ & \qquad\qquad \Box[\neg Q]_v \land \Box(\mathbb{E}\langle Q\rangle_v \Rightarrow \mathbb{E}\langle A_i\rangle_v))) \\ & \Rightarrow\ (\mathrm{XF}_v(Q) \equiv \forall\, i \in I : \mathrm{XF}_v(A_i)) \end{aligned}$$

It is perhaps interesting, but of no practical significance, that the theorem is valid even if the set of actions A_i is uncountably infinite.

The completeness result for safety properties in Theorem 4.1 can be extended as follows to arbitrary properties. A proof is sketched in the Appendix.

Theorem 4.8 Let $\mathbf{x}$ be the list $x_1, \ldots, x_n$ of variables and let F be a property such that $F(\sigma)$ depends only on the values of the variables $\mathbf{x}$ in σ, for any behavior σ. There exists a formula S equal to $\mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathbf{x},y\rangle} \land \mathrm{WF}_{\langle \mathbf{x},y\rangle}(\mathit{Next})$, where *Init* and *Next* are defined in terms of F, y is a variable not among the variables $\mathbf{x}$, and the variables of S are $\mathbf{x}$ and y, such that $\models F \Rightarrow G$ iff $\models [\![S]\!] \Rightarrow G$, for any property G. If F is a safety property, then the conjunct $\mathrm{WF}_{\langle \mathbf{x},y\rangle}(\mathit{Next})$ is not needed.

Like Theorem 4.1, this result is of theoretical interest only.

4.2.8 What is Fairness?

Before TLA, concurrent abstract programs were generally written in something like a coding language. Fairness meant that each process had to execute its next atomic statement when it could. Viewed in terms of TLA, each atomic statement was described by an action, and fairness meant fairness of those actions. Usually that meant weak fairness of the action, but when the statement was a synchronization primitive, it sometimes meant strong fairness. For rigorous reasoning, those fairness requirements were expressed as requirements on when control in the process had to move from one control point to another [44].

With TLA, fairness was generalized to weak and strong fairness of arbitrary actions. We have considered a fairness property for a safety property S to be a formula L that is the conjunction of weak and strong fairness conditions on actions such that $\langle S, L\rangle$ is machine closed. However, weak and strong fairness of an action are defined in terms of how the action is written, not in terms of its semantics. While we have given semantic definitions of safety, liveness, and machine closure; we have not done it for fairness.

I only recently learned that a semantic definition of fairness was published in 2012 by Völzer and Varacca [48]. Their definition of what it means for a property L to be a fairness property for a safety property S can be stated in terms of the following infinite two-player game. Starting with *seq* equal to the empty sequence, the two players forever alternately take steps that append a finite number of states to *seq*. The only requirement on the steps is that after each one, *seq* must satisfy S. The second player wins the game if she makes *seq* an infinite sequence that satisfies L. (Since S is a safety property, *seq* must satisfy S.) They defined L to be a fairness property for S iff the second player can always win, regardless of what the first player does (as long as he follows the rules).

It is mathematically meaningless to say that a definition is correct. However, this seems to be the only reasonable definition that includes weak and strong fairness such that fairness implies machine closure and the conjunction of countably many fairness properties is a fairness property. This definition also encompasses other fairness properties that have been proposed, including one called hyperfairness [33].

I believe that weak and strong fairness of actions are the only fairness properties that are relevant to abstract programs. However, this general definition is interesting because it provides another way to think about fairness. More importantly, it's interesting because concepts we are led to by mathematics often turn out to be useful.

4.3 Possibility and Accuracy

4.3.1 Possibility Conditions

Informally, a safety property states what an abstract program is allowed to do and a liveness property states what it must do. If a behavior violates a safety property, then it does so at a particular step in the behavior. Therefore, we can also view a safety property as stating what a program must not do—that is, it must not take a step that violates the property. So, liveness says what must happen and safety says what must not happen. That a program eventually sets x to 0 is a liveness property, that it never sets x to 0 is a safety property.

Possibility says what *might* happen. That a program might set x to 0 is a possibility condition. It is not a property, because it is not a predicate on behaviors. It is satisfied by an abstract program iff there is some behavior of the program in which x is set to 0. We can tell that the program satisfies it if we see such a behavior. But seeing one behavior that doesn't satisfy it doesn't tell us whether or not some other behavior might satisfy it. However, we will see that possibility conditions can be expressed as properties that explicitly mention the program's actions.

Knowing that something might be true of a system, but knowing nothing about the probability of its being true, is of almost[8] no practical use. The only way I know of calculating such probabilities is to view the abstract program as a state-transition system, attach probabilities to the various transitions, and mathematically analyze that system—for example, using Markov analysis. Usually, the state-transition system would be a more abstract program implemented by the program of interest.

While possibility conditions of systems are of little interest, we don't reason about systems; we reason about abstract programs that describe systems. Verifying that a program satisfies a possibility condition can be a way of checking what we will call here the *accuracy* of an abstract program—that it accurately describes the system it is supposed to describe. For example, if the system doesn't control when users send it input, a program that accurately describes the system and its users should satisfy the condition that it's always possible for users to enter input.

4.3.2 Expressing Possibility in TLA

There exist logics for expressing possibility properties and tools for checking them. Such tools could be built to check those properties for abstract TLA programs, and it might be worthwhile to do so. But we will see how that can be avoided.

Even though a possibility condition is not a property, that an abstract program satisfies a possibility condition can be expressed by a TLA formula that depends on the program and the possibility condition. For example, suppose S is a TLA description of an abstract program and the action *Input* is a subaction of its next-state action that describes users entering input. That it is always possible for users to enter input could be considered to mean that the *Input* action is enabled in every reachable state of the program, which is asserted by

$$\models S \;\Rightarrow\; \Box\, \mathbb{E}(\mathit{Input})$$

However, "always possible" might instead mean that from any reachable state, there is a sequence of possible steps that reach a state with $\mathbb{E}(\mathit{Input})$ true—a condition we will call "always eventually possible". To express this

[8]Section 7.1 explains one way in which a possibility condition can be used to verify a property of a system and could therefore be of practical use.

and other possibility conditions in TLA, we can use the action composition operator defined in Section 3.4.1.4. Recall that for any action A, the action A^+ is true of a step $s \to t$ iff t is reachable from s by a sequence of one or more A steps.

Now consider an abstract program $Init \wedge \Box[Next]_v$ where $Init$ is a state predicate, $Next$ is an action, and v is the tuple of all variables that appear in $Init$ or $Next$. Let's abbreviate $([Next]_v)^+$ as $[Next]_v^+$. If s is a reachable state of the program, then $s \to t$ is a $[Next]_v^+$ step iff it is possible for an execution of the program to go from state s to state t. (Since $[Next]_v$ allows stuttering steps, t can equal s. In fact, $[Next]_v^+$ is equivalent to $[Next^+]_v$.) A state t is a reachable state of the program iff there is a state s satisfying $Init$ such that $s \to t$ is a $[Next]_v^+$ step. In other words, t is a reachable state of the program iff there is a state s such that $s \to t$ is an $Init \wedge [Next]_v^+$ step.

We can now express the condition that it is always eventually possible for the user to enter input, meaning that from any reachable state, it is possible to reach a state in which $\mathbb{E}(Input)$ is true. We generalize this condition by replacing $\mathbb{E}(Input)$ with an arbitrary state predicate P. For the abstract program $Init \wedge \Box[Next]_v$, that P is always eventually possible is expressed as:

$$(4.26) \quad \models Init \wedge \Box[Next]_v \;\Rightarrow\; \Box\, \mathbb{E}([Next]_v^+ \wedge P')$$

This example indicates that it should be possible to express possibility conditions in TLA using $Next^+$ and the $\mathbb{E}$ operator. However, like (4.26), the resulting TLA formulas are quite different from the ones that arise in checking that an abstract program satisfies a property. Different tools would be needed to verify that a program satisfies a possibility condition expressed in this way. It would be nice to be able to verify possibility conditions by verifying the same kind of properties that arise in verifying that an abstract program satisfies a liveness property. Here is how it can be done for the condition that it is always possible to reach a state satisfying P.

This condition obviously holds if the program satisfies the property that in any reachable state, a state satisfying P *must* eventually occur—that is, if the program satisfiers the property $\Box\Diamond P$. Let S equal $Init \wedge \Box[Next]_v$. The safety property S will not imply the liveness property $\Box\Diamond P$ unless P is true in all reachable states of F—that is, unless S implies $\Box P$. However, if F is a fairness property for S, so $\langle S, F \rangle$ is machine closed, then $S \wedge F$ has the same set of reachable states as S. So, any state satisfying P can be reached from a reachable state of S iff it can be reached from a state satisfying $S \wedge F$. Therefore, it suffices to verify that a state satisfying P

can be reached from every reachable state of $S \wedge F$, which is true if $S \wedge F$ implies $\Box\Diamond P$. Therefore, we can verify that P is always eventually possible by verifying

$$\models \mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge F \;\Rightarrow\; \Box\Diamond P$$

for a fairness property F of $\mathit{Init} \wedge \Box[\mathit{Next}]_v$. By Theorem 4.6, we can ensure that F is a fairness property for S by writing it as the conjunction of strong fairness properties. (Since strong fairness implies weak fairness, there is no need to use weak fairness properties.) There is a completeness result that essentially says that such a fairness property always exists for any state predicate P if S has the standard form $\mathit{Init} \wedge \Box[\mathit{Next}]_v$ [32].

I suspect that TLA can in a similar way express possibility for a more general class of possibility conditions than the two possible interpretations of "always possible". However, the only other possibility condition I have found to be useful for checking the accuracy of an abstract program is a very simple one, discussed in the following section.

4.3.3 Checking Accuracy

Using TLA to check that a state predicate P is always eventually possible may not be easy, since it requires finding a fairness condition that implies P is true infinitely often. There's a simpler condition that is easier to check: it's possible for P to be true (at least once). It would be more useful to check the stronger condition of always eventually possible. However, I have found that most people, including me, don't spend enough time checking the accuracy of their abstract programs. The weaker check is likely to be more helpful in practice because it's more likely to be done than one that requires more effort.

It's possible for P to be true means that it is true in some reachable state. This is equivalent to the assertion that $\neg P$ is not true in all reachable states—in other words, that $\neg P$ is not an invariant of the program. We can check this by asking a tool for checking invariance to check if $\neg P$ is an invariant. If the tool reports that it isn't, then it's possible for P to be true. For example, if a model checker reports that your mutual exclusion algorithm satisfies mutual exclusion, you should check that it's possible for a process to enter its critical section. This is especially true if you did not have to make many corrections to reach that point. Remember that a program that takes no non-stuttering steps satisfies most safety properties.

Tools can provide other ways of checking the accuracy of a program. For example, if *Input* is a subaction of the program's next-state action, a TLA$^+$

model checker called TLC reports how many different steps satisfying *Input* occur in behaviors of the program. If it finds no such steps, then it is not always possible for an *Input* step to occur with either definition of "always possible". Finding too few such steps can also be an indication that the program is not accurate.

Accuracy of an abstract program cannot be formally defined. It means that a program really is correct if it implements the abstract program. In other words, an abstract program is accurate iff it means what we want it to mean, and our desires can't be formally defined. That accuracy can't be formally defined does not mean it's unimportant. There are quite a few important aspects of programs that lie outside the scope of our science of correctness.

4.4 Real-Time Programs

Real-time abstract programs are ones in which timing constraints ensure that safety properties hold. Real time is most often used in concrete programs to ensure not safety but liveness. It appears in timeouts that guarantee something eventually happens. For example, to guarantee that a message is eventually delivered despite possible message loss, a timeout occurs if an acknowledgement of the message is not received soon enough after it is sent, and the message is then resent. A program that uses timeouts only in this way is not a real-time program. The actual time at which a timeout occurs affects only performance, not correctness. Therefore, timeout can be modeled in an abstract program as an event that must eventually occur but can occur at any time. The program can use liveness to abstract away time. We need to write a real-time abstract program only if timing constraints are used to ensure safety properties.

Scientists have been dealing mathematically with real-time systems for centuries by simply representing time as the value of a variable. It has been known for decades that this works for real-time programs too [6]. I will illustrate how it is done with a mutual exclusion algorithm of Michael Fischer.[9]

I believe that most work on the correctness of real-time programs has considered only safety properties. Instead of requiring that something eventually happens, it requires the stronger property that it happens within some fixed amount of time, which is a safety property. Fischer's Algorithm

[9]Fischer sent this algorithm in an email to me [12]. I believe Martín Abadi and I were the first to describe it in print [1].

is more general because in addition to using real-time to ensure mutual exclusion, a safety property, it uses fairness to ensure deadlock freedom, a liveness property.

In the past 40 years, I have had essentially no contact with engineers who build real-time systems. I know of only one case in which TLA was used to check correctness of the design of a commercial real-time system [5]. From the point of view of our science, there is nothing special about real-time programs. However, how well tools work can depend on the application domain. The TLA$^+$ tools were not developed with real-time programs in mind, and it's unclear how useful they are in that domain.

4.4.1 Fischer's Algorithm

The algorithm without its timing constraints is described by the pseudocode in Figure 4.7. The constant *Procs* is the set of processes, and *none* is some constant that is not in *Procs*. The global variable x is read and written by all the processes and, as usual, the value of $pc(p)$ is the label of the next statement to be executed by process p. As explained in Section 4.2.2.1, the statement **await** P is a synchronization primitive that allows the program to continue only when the state predicate P is true. Thus, the *wait* statement of process p is described by the action:

$$\begin{array}{l} \land\ pc(p) = wait \\ \land\ x = none \\ \land\ pc' = (pc \text{ EXCEPT } p \mapsto w1) \\ \land\ x' = x \end{array}$$

With no time constraints, mutual exclusion is easily violated. Two processes can execute the *wait* statement when x equals *none*, then statements $w1$ and $w2$ can both be executed by the first process and then by the second one, putting both processes in the critical section. Mutual exclusion is ensured by timing constraints.

We assume that each step is executed instantaneously at a certain time, and that each process executes $w1$ at most δ seconds after it executes *wait* and executes $w2$ at least ϵ seconds after it executes $w1$, for constants δ and ϵ with $\delta < \epsilon$. (The algorithm doesn't specify what the time units are; we will call them seconds for convenience.) It's a nice exercise to show that this ensures mutual exclusion by assuming that two processes are in their critical sections and showing that the necessary reads and writes of x that allowed them both to enter the critical section must have occurred in an order that violates the timing constraints if $\delta < \epsilon$. While it may be good enough for

```
variables $x = none$ ;
process $p \in Procs$
   variables $pc = ncs$ ;
   while TRUE do
      ncs:   skip ;   noncritical section
      wait:  await $x = none$ ;
      w1:    $x := p$ ;
      w2:    if $x \neq p$ then goto wait end if ;
      cs:    skip ;   critical section
      exit:  $x := none$
   end while
end process
```

Figure 4.7: Fischer's Algorithm.

such a simple algorithm, this kind of behavioral reasoning is unreliable for more complicated programs.

To verify mutual exclusion more rigorously, we describe Fischer's Algorithm with its timing constraints as an abstract program. This requires adding a variable whose value represents the current time. Scientists usually call that variable t, but I like to call it *now*. We also add a variable *rt*, where the value of $rt(p)$ records the time at which certain actions of process p were executed. The program also contains an additional process called *Time* that advances time.

The algorithm is written in pseudocode in Figure 4.8. The initial value of *now* can be any number in the set $\mathbb{R}$ of real numbers, and the initial value of *rt* can be any function from the set of processes to $\mathbb{R}$. Let's now examine the code of process p. Two assignments to $rt(p)$ have been added. Each sets $rt(p)$ to the time at which the action in which it appears is executed, the actions being the ones performed when the program is at control points *wait* and $w1$. Also added is an **await** statement at $w2$ that allows the action to be performed only when $now - rt(p) \geq \epsilon$. This **await** enforces the requirement that the $w2$ action must not be executed until at least ϵ seconds after execution of the $w1$ action.

Let's now examine the *Time* process. It repeatedly performs a single atomic action, so it has just one control point that needs no label. That action increases the value of *now*. The $:\in$ operation is like the assignment operation $:=$ except with $=$ replaced by $\in$. That is, a step of the *Time* process's action assigns to *now* an arbitrary element of the set on the right-

```
variables $x = none$, $now \in \mathbb{R}$, $rt \in (Procs \to \mathbb{R})$ ;
process $p \in Procs$
  variables $pc = ncs$ ;
  while TRUE do
    $ncs$:  skip ;   noncritical section
    $wait$: await $x = none$ ;
            $rt(p) := now$ ;
    $w1$:   $x := p$ ;
            $rt(p) := now$ ;
    $w2$:   await $now - rt(p) \geq \epsilon$ ;
            if $x \neq p$ then goto $wait$ end if ;
    $cs$:   skip ;   critical section
    $exit$: $x := none$
  end while
end process
process $Time$
  while TRUE do
    $now :\in \{t \in \mathbb{R}$ :
                $\wedge\ t > now$
                $\wedge\ \forall\, p \in Procs : (pc(p) = w1) \Rightarrow (t \leq rt(p) + \delta)\,\}$
  end while
end process
```

Figure 4.8: Fischer's Algorithm with explicit time.

hand side of the $:\in$. The action can assign to *now* any value t greater than its current value subject to the condition that $t \leq rt(p) + \delta$ for every process p at control point $w1$. It is this condition that enforces the requirement that a process must execute statement $w1$ within δ seconds of when it executes the *wait* statement.

Fischer's Algorithm illustrates the basic method of representing real-time constraints in an abstract program. Lower bounds on how long it must take to do something are described by enabling conditions on the algorithm's actions. Upper bounds are described by enabling conditions on the action that advances time. There are a number of ways of enforcing these bounds. The use of the variable *rt* in Fischer's algorithm shows one way. Another is to use variables whose values are the number of seconds remaining before an action must be executed (lower bounds) or can be executed (upper bounds)—variables whose values are decremented by the time-advancing action.

The idea of an abstract program constraining the advance of time is mind-boggling to most people, since they view a program as a set of instructions. They see it as the program stopping time. You should by now realize that an abstract program is a description, not a set of instructions. It describes a universe in which the algorithm is behaving correctly. That description may constrain the algorithm's environment, which is the part of the universe that the algorithm doesn't control—for example, its users. Time is an important part of that environment if the amount of time it takes to perform the algorithm's actions is relevant to its correctness.

4.4.2 Correctness of Fischer's Algorithm

Having written Fischer's algorithm as an abstract program, we know how to verify its correctness. Mutual exclusion is an invariance property, and to understand why the algorithm satisfies it we need to find the inductive invariant that explains why the algorithm satisfies that property. As usual, the inductive invariant asserts type-correctness of all the variables. The interesting part of the invariant makes the following assertions about each process p:

- If control is at $w1$, then the current time is at most δ seconds after the time at which p just executed the *wait* statement.
- If control is at *cs* or *exit*, then $x = p$ and in no process is control at $w1$. (This condition implies mutual exclusion.)
- If p is at $w2$ and $x = p$, then any process with control at $w1$ must execute statement $w1$ before p can execute statement $w2$.

These assertions about every process p are expressed mathematically as:

$$
\begin{array}{l}
\forall\, p \in \mathit{Procs} \;: \\
\quad \land\;\; (pc(p) = w1) \Rightarrow (rt(p) \leq \mathit{now} \leq (rt(p) + \delta)) \\
\quad \land\;\; (pc(p) \in \{\mathit{cs}, \mathit{exit}\}) \;\Rightarrow\; (x = p) \,\land\, (\forall\, q \in \mathit{Procs} \;:\; pc(q) \neq w1) \\
\quad \land\;\; (pc(p) = w2) \,\land\, (x = p) \;\Rightarrow \\
\qquad\qquad \forall\, j \in \mathit{Procs} \;:\; (pc(q) = w1) \;\Rightarrow\; ((rt(q) + \delta) < (rt(p) + \epsilon))
\end{array}
$$

You should understand why the three conjuncts in this formula are the three assertions expressed informally above. Adding the type-correctness part and proving that it is an inductive invariant is a good exercise if you want to learn how to write proofs.

Under suitable fairness assumptions, Fischer's Algorithm is deadlock free. Recall that deadlock freedom for a mutual exclusion algorithm means it's always true that if some process is trying to enter the critical section, then some process (not necessarily the same process) will eventually do so. Deadlock freedom of Fischer's Algorithm follows from the algorithm having this additional invariant:

(4.27) $(x \neq \textit{none}) \;\Rightarrow\; (pc(x) \in \{w2, cs, \textit{exit}\})$

Here's a sketch of a proof by contradiction that the algorithm is deadlock free. Suppose some process is at *wait* and no process is ever in its critical section. Eventually, some set of processes will be forever in their noncritical sections, and one or more processes will forever have control at *wait*, $w1$, or $w2$. Eventually the latter processes will all wind up waiting at the *wait* statement with $x \neq \textit{none}$. But that contradicts the invariant (4.27), which implies that process x cannot be at *wait*.

4.4.3 Fairness and Zeno Behaviors

What fairness requirements of the abstract program of Figure 4.8 are assumed in the informal argument that the program is deadlock free? If $\textit{procStep}(p)$ is the next-state action of a process p in *Procs*, then we naturally assume weak fairness of the action $\textit{procStep}(p) \wedge (pc \neq \textit{ncs})$. What about fairness of the *Time* process? Let's call that process's action *timeStep*. The obvious choice is to let it be strong fairness of *timeStep*. However, that allows the following behavior: While all other processes in *Procs* remain in their noncritical sections, a process p executes the *wait* statement and then, at time t, executes statement $w1$ that sets $rt(p)$ to t. Repeated executions of action *timeStep* then set *now* to $t + \epsilon/2$, then $t + 2 * \epsilon/3$, then $t + 3 * \epsilon/4$, and so on. Process p must wait forever at $w2$ because *now* is always less than $t + \epsilon$ and the $w2$ action is enabled only when $\textit{now} \geq t + \epsilon$. Such a behavior, in which time remains bounded, is called a Zeno behavior.

The most natural way to avoid the problem of Zeno behaviors is to make the abstract program describing Fischer's Algorithm disallow them. The obvious way to do that is to conjoin this liveness property:

(4.28) $\forall\, t \in \mathbb{R} : \Diamond(\textit{now} > t)$

which asserts that the value of time is unbounded. However, this isn't necessarily a fairness property. It's easy to write an abstract program that allows only Zeno behaviors, so conjoining the liveness property (4.28) produces a program that allows no behaviors. For example, we can add timing constraints to the program of Figure 4.7 that require a process both to execute statement $w1$ within δ seconds after executing statement *wait* and to wait at least ϵ seconds after executing *wait* before executing $w1$, with $\delta < \epsilon$. If a process executes *wait* at time t, then $now \leq t + \delta$ must remain true forever. If we added fairness properties that required processes eventually to reach the *wait* statement and execute it if it's enabled, then the program would allow only Zeno behaviors.

We can ensure that Fischer's Algorithm satisfies (4.28) by having it require an appropriate fairness condition on the advancing of time. The condition we need is strong fairness of the action $timeStep \wedge (now' = exp)$, where exp is the largest value of now' permitted by the values of $rt(p)$ for processes p with control at $w1$, or $now + 1$ if there is no such process. More precisely:

$$\begin{array}{lll} exp & \triangleq & \text{LET } T \triangleq \{rt(p) + \delta : p \in \{q \in Procs : pc(q) = w1\}\} \\ & & \text{IN } \quad \text{IF } T = \{\} \text{ THEN } now + 1 \text{ ELSE } Min(T) \end{array}$$

where $Min(T)$ is the minimum of the nonempty set T of real numbers. With this fairness condition on advancing time and the conjunction of the fairness conditions for the processes in *Proc*, Fischer's Algorithm satisfies (4.28) and the proof sketch that the algorithm is deadlock free can be made rigorous.

If we are interested only in safety properties, there is no need for an abstract program to rule out Zeno behaviors. A program satisfies a safety property iff all finite behaviors allowed by the program satisfy it, and a Zeno behavior is an infinite behavior. In many real-time programs, liveness properties are of no interest. Correctness means not that something eventually happens but that it happens within a certain length of time, which is a safety property. Zeno behaviors then make no difference, and there is no reason to disallow them.

Even if Zeno behaviors don't matter, the absence of non-Zeno behaviors can be a problem. Since real time really does increase without a bound, an abstract program in which it is not always possible for time to become arbitrarily large is unlikely to be accurate. Therefore, we almost always want to ensure that a real-time program satisfies the condition that for any $t \in \mathbb{R}$, it is always possible for $now > t$ to be true. This is true iff, for any $t \in \mathbb{R}$, from any reachable state of the program it is always possible for $now > t$ to be true. This is the kind of possibility condition considered in Section 4.3. We saw there that if the program *Safe* is a safety property that satisfies this condition, then we can verify that it does so by finding a conjunction F of fairness properties for *Safe* and verifying:

(4.29) $\models Safe \wedge F \Rightarrow \Box\Diamond(now > t)$

for all $t \in \mathbb{R}$. (Since a real-time program never allows *now* to decrease, it suffices to verify that *Safe* $\wedge F$ implies (4.28).)

4.4.4 Discrete Time

Verifying properties of real-time programs is easier if we assume time is discrete and *now* always equals a multiple of some time unit. It may seem obvious that, since concrete programs run on real computers reading the current time from a clock that advances in discrete steps, we can always assume discrete time. However, different processes can be executed on different computers whose clocks can run at slightly different rates. Still, it seems likely that an abstract program will be sufficiently accurate if it assumes time changes only in one yoctosecond (10^{-24} second) increments. So, in practice we should be able to assume that the values of *now* and any time constants are integers, leaving unspecified how long one time unit is.

When writing proofs, there doesn't seem to be much reason to use discrete time. The main advantage of discrete time is that tools for automatically verifying properties of ordinary abstract programs can, in principle, handle discrete real-time programs. For example, I didn't prove that what I claimed in Section 4.4.2 to be an inductive invariant of Fischer's Algorithm actually is one. Instead, I used a model checker to check that it is, which gave me enough confidence to make the claim.

Many model checkers are based on enumerating reachable states, usually on a small instance of the program—for example, with a small number of processes. This is impossible with continuous time, in which a single state can have possible next states with uncountably many values of *now*. There are still infinitely many reachable states with discrete time because the values of *now* are unbounded, but counterexamples to incorrect safety properties and to (4.28) for a particular time t can be found by examining all reachable states with *now* less than some value.

The number of reachable states that must be examined can be reduced for real-time programs that satisfy a condition called *symmetry under time translation*. This condition asserts that for every $d \in R$ there is a *time-translation* mapping T_d from states to states such that: For every state s, the value of *now* in state $T_d(s)$ equals d plus its value in s, and any step $s \rightarrow t$ satisfies the program's next-state action iff $T_d(s) \rightarrow T_d(t)$ does. For example, Fischer's Algorithm is symmetric under the time translations T_d defined by letting the values of *now* and $rt(p)$ in $T_d(s)$ equal d plus their values in s, and letting the values of x and pc be the same in s and $T_d(s)$.

Suppose S is a program symmetric under the time-translation functions

T_d, and for simplicity assume that the program's initial predicate asserts $now = 0$. Let's call states s and t *translation equivalent* iff $t = T_d(s)$ for some d. If P is a safety property containing only variables whose values are left unchanged by the functions T_d, then to verify that P is satisfied by S we can verify that it is satisfied by the program $\widehat{S}$ obtained from S by considering two translation equivalent states to be the same state. Often, there will be some time λ such that every reachable state of S is translation equivalent to a state in which $now \leq \lambda$, in which case $now \leq \lambda$ in every state of $\widehat{S}$. This implies we can verify that S satisfies (4.28) by checking that it satisfies $\Diamond(now \geq \lambda + 1)$, which requires examining only reachable states with $now \leq \lambda + 1$. The details can be found elsewhere [35].

Being able to reduce verification of a discrete-time program to examining reachable states with *now* less than some value would still leave an enormous number of states to consider if *now* advanced in yoctosecond steps. Henzinger, Manna, and Pnueli [17] proved that for a class of programs called timed transition systems, certain properties can be verified with discrete time in which *now* advances only in reasonably sized steps. Timed transition systems are essentially programs in which, like Fischer's Algorithm, time is used only to require minimum and maximum delays between when a program action becomes enabled and when it either must be executed (maximum delay) or may be executed (minimum delay). If those delays are constants that are all multiples of some time unit Δ, then a certain class of properties can be verified by letting time always be a multiple of Δ. That class of properties are ones in which replacing each state in a behavior by time-translating it by d with $-\Delta < d \leq 0$ does not change whether the behavior satisfies the property. It includes properties that depend only on variables whose values are unchanged by time translation. It also includes the property (4.28).

One reason that has been given for preferring continuous time is that it is necessary for composing programs. It would be difficult to compose a program in which a clock tick represents a nanosecond with one in which a clock tick represents a millisecond. However, we can easily describe a program in terms of a clock that ticks at an unspecified rate and an unspecified constant that equals the number of clock ticks in a second.

Program composition is discussed in Section 7.2, where it is shown how the composition of two abstract programs can be represented as the conjunction of the TLA formulas that describe them. Moreover, a real-time abstract program can be written as the conjunction of two formulas, one describing an ordinary, untimed program, the other specifying the required timing constraints [1]. However, this kind of abstract program composition is not yet usable in practice.

4.4.5 Hybrid Systems

A hybrid computer system is one that controls physical processes—for example, one that flies an airplane or runs a chemical plant. An abstract program that describes such a system is a real-time program that describes not only the passage of time but also other physical quantities like altitude or pressure. There is no fundamental difference between programs that describe hybrid systems and other real-time programs. The current altitude or pressure is represented by a variable like any other variable. Abstract programs describing hybrid systems differ from other real-time abstract programs only in the math used to describe them. If the variable *prs* describes the current pressure, then the time-advancing subaction of the next-state action might contain a subformula like:

$$(4.30)\quad prs' = prs + \int_{now}^{now'} exp\, dt$$

for some expression *exp* containing the bound variable *t* and other variables [30].

It may seem that a representation of the behavior of a continuous process by a sequence of discrete states would not be sufficiently accurate. For example, if it is required that the pressure not be too high, violation of that requirement would not be found if it occurred during the time between two successive states of the behavior. This is not a problem because correctness means that a property is true of all possible behaviors, and the possibility of the pressure being too high at some time is revealed by a behavior containing a state in which *now* equals that time.

Other than the differences implied by the use of continuous math, such as the calculus in (4.30), rather than discrete math, proving properties of hybrid programs is the same as proving properties of other real-time abstract programs. Automatic tools like model checkers for ordinary abstract programs seem to be unsuitable for checking abstract programs in which variables represent continuously varying quantities. Methods have been developed for checking such programs [11].

Chapter 5

Refinement

We have discussed one abstract program implementing another. We now consider more carefully what that means. We write abstract programs with TLA formulas, and it is rather weird to talk about one formula implementing another. Computer scientists who view programs mathematically generally use the term *refinement* rather than *implementation*. Henceforth, we will use the two terms interchangeably. There are two aspects to refinement:

Step Refinement The refining program has a finer grain of atomicity. This means that a non-stuttering step of the high-level program can correspond to multiple steps of the refining program, all but one of them implementing stuttering steps of the high-level program. In the example of Section 3.5.1, the hour-minute-second clock *HMS* refines the hour-minute clock *HM*. Every non-stuttering step of *HM*, which advances the minute, corresponds to 60 steps of *HMS*, each changing the second and one of them changing the minute.

Data Refinement A program refining another program can also refine the representation of data used by the higher-level program. This will be illustrated by refining a higher-level program that uses numbers with a program that implements a number by a sequence of digits.

Refinement usually involves both step and data refinement, with step refinement manifest as operations on the lower-level data requiring more non-stuttering steps than the corresponding operations on the higher-level program's data.

5.1 A Sequential Algorithm

In step refinement, the additional steps taken by the lower-level program correspond to stuttering steps in the higher-level one. We consider an extreme example of this: a program that terminates after taking a single non-stuttering step that is refined by a traditional sequential program that computes a value and stops.

This example is used because it's simple and nicely illustrates data refinement. Its use does not imply that this way of looking at refinement is the best one to use for traditional programs. There are other methods for reasoning about refinement of traditional programs that are probably better than our science of more general programs [20]. We consider only safety in this example; liveness is straightforward.

The high-level abstract program *Add* begins with variables x and y equal to arbitrary natural numbers, sets the variable z to their sum, and terminates. Termination is indicated by changing the value of a Boolean-valued variable *end* to TRUE. Here is the definition:

$$\begin{aligned}
&\mathit{InitA} && \triangleq\ (\mathit{end} = \textsc{false}) \land (x \in \mathbb{N}) \land (y \in \mathbb{N}) \\
&\mathit{NextA} && \triangleq\ \neg \mathit{end} \land (z' = x + y) \land (\mathit{end}' = \textsc{true}) \\
&\mathit{vA} && \triangleq\ \langle x, y, z, \mathit{end} \rangle \\
&\mathit{Add} && \triangleq\ \mathit{InitA} \land \Box[\mathit{NextA}]_{\mathit{vA}}
\end{aligned}$$

Note that *Init* does not specify the value of z. Its initial value doesn't matter. Note also that action *NextA* does not specify the values of x' or y'. You should realize by now that this doesn't mean those values are unchanged; it means that their new values are unspecified. We are assuming that the final values of x and y don't matter; we care only about the final value of z.

5.1.1 A One-Step Program

We refine *Add* by a program *AddS* in which a natural number is represented by a finite ordinal sequence of decimal digits—that is, by an element of the set $\mathit{Seq}(0\,..\,9)$. For convenience, we number the digits from right to left, so the sequence $\langle 1, 2, 3 \rangle$ represents the number 321. Thus a sequence *seq* of digits represents the number $\mathit{Val}(\mathit{seq})$ defined as follows, where the empty sequence is defined to represent 0.

$$\begin{aligned}
\mathit{Val}(\mathit{seq})\ \triangleq\ \ \textsc{if}\ \ \mathit{seq} = \langle\,\rangle\ \ &\textsc{then}\ \ 0 \\
&\textsc{else}\ \ \mathit{seq}(1) + 10 * \mathit{Val}(\mathit{Tail}(\mathit{seq}))
\end{aligned}$$

Let $\oplus$ represent addition of numbers represented in this way as sequences of digits. In other words, $\oplus$ satisfies

$$Val(s \oplus t) = Val(s) + Val(t)$$

for all sequences s and t of digits. We would expect *Add* to be refined by the program *AddS* obtained by replacing + by $\oplus$ and $\mathbb{N}$ by $Seq(0\,..\,9)$ in the definition of *Add*. To avoid confusing the variables of the two programs, we'll also replace x, y, z, and *end* by u, v, w, and *fin*, so *AddS* is defined by:

$$\begin{array}{lcl} InitS & \triangleq & (fin = \text{FALSE}) \land (u \in Seq(0\,..\,9)) \land (v \in Seq(0\,..\,9)) \\ NextS & \triangleq & \neg fin \land (w' = u \oplus v) \land (fin' = \text{TRUE}) \\ vS & \triangleq & \langle u, v, w, fin \rangle \\ AddS & \triangleq & InitS \land \Box[NextS]_{vS} \end{array}$$

Exactly what does it mean for *AddS* to refine *Add*? I believe the natural definition is: If we look at any behavior of *AddS* and interpret the numbers represented by the sequences u, v, and w of digits to be the values of x, y, and z and we interpret the value of *fin* to be the value of *end*, then we get a behavior of *Add*. More precisely, let "$\leftarrow$" mean "is represented by". That *AddS* refines *Add* means that a behavior satisfying *AddS* represents a behavior satisfying *Add* with this representation of the variables of *Add* in terms of the variables of *AddS*:

(5.1) $x \leftarrow Val(u) \quad y \leftarrow Val(v) \quad z \leftarrow Val(w) \quad end \leftarrow fin$

Here's an example to illustrate this, where the first two-state sequence is a finite behavior satisfying *AddS* and the second two-state sequence is the finite behavior it represents. Remember that *AddS* leaves unspecified the value of w in an initial state and the values of u and v in a halting state. A "?" in the second behavior means the value is unspecified because, as explained in Section 2.2.7, $Val(seq)$ is a meaningless expression if *seq* isn't a sequence of numbers.

$$\begin{bmatrix} u & :: & \langle 1,2,3 \rangle \\ v & :: & \langle 3,2 \rangle \\ w & :: & \sqrt{2} \\ fin & :: & \text{FALSE} \end{bmatrix}_0 \rightarrow \begin{bmatrix} u & :: & \langle 5 \rangle \\ v & :: & -27 \\ w & :: & \langle 4,4,3 \rangle \\ fin & :: & \text{TRUE} \end{bmatrix}_1$$

$$\begin{bmatrix} x & :: & 321 \\ y & :: & 23 \\ z & :: & ? \\ end & :: & \text{FALSE} \end{bmatrix}_0 \rightarrow \begin{bmatrix} x & :: & 5 \\ y & :: & ? \\ z & :: & 344 \\ end & :: & \text{TRUE} \end{bmatrix}_1$$

As you can see, the second finite behavior satisfies *Add*, since $321+23$ equals 344 and *end* has the values the abstract program *Add* says it should.

Let's look closely at this example. What it shows is that when we perform the substitutions (5.1) for the variables of *Add* in a behavior satisfying formula *AddS*, we get a behavior that satisfies formula *Add*. In other words, in any behavior: if the behavior satisfies *AddS*, then it satisfies the formula *Add* when we perform the substitutions (5.1). This means that the following formula is true:

$$\text{(5.2)}\quad \models \mathit{AddS} \;\Rightarrow\; (\mathit{Add}\ \textsc{with}\ x \leftarrow \mathit{Val}(u),\ y \leftarrow \mathit{Val}(v),\ z \leftarrow \mathit{Val}(w),\ \mathit{end} \leftarrow \mathit{fin})$$

This is what it means for *AddS* to refine *Add* under the representation defined by (5.1). That representation is called a *refinement mapping.* Formula (5.2) asserts that *AddS* implements *Add* under this refinement mapping.

I find (5.2) beautiful. We've already seen that, viewed in terms of TLA, step refinement is implication. Now we see that data refinement is substitution—the ordinary mathematical operation of substituting expressions for variables in a formula. How beautifully simple! In science, beauty is not an end in itself. It's a sign that we're doing something right.

5.1.2 Two Views of Refinement Mappings

There are two ways to view the refinement mapping (5.1) that appears in (5.2). To understand them, let's simplify things by ignoring irrelevant variables and letting *state* mean program state—an assignment of values to the program's variables. Let an S-state be a state of *AddS*, which is an assignment of values to the variables u, v, w and *end* of *AddS*; and let an A-state be an assignment of values to the variables x, y, z, and *end* of *Add*. Let an S-behavior or A-behavior be a sequence of S-states or A-states, respectively.

The first way to view the refinement mapping is as a mapping f that maps S-states to A-states. We can define the mapping $\widehat{f}$ from S-behaviors to A-behaviors in terms of f by $\widehat{f}(\sigma)(i) \triangleq f(\sigma(i))$ for any S-behavior σ. That *AddS* implements *Add* under the refinement mapping f means $[\![\mathit{AddS}]\!](\sigma) \Rightarrow [\![\mathit{Add}]\!](\widehat{f}(\sigma))$ for every S-behavior σ.

The second way to view refinement is expressed in (5.2), which states that *AddS* implements *Add* under the refinement mapping means $[\![\mathit{AddS}]\!](\sigma) \Rightarrow [\![\mathit{Add}\ \textsc{with}\ \ldots]\!](\sigma)$ for every S-behavior σ.

In the first view, the refinement mapping maps low-level behaviors to high-level behaviors—that is, behaviors of *AddS* to behaviors of *Add*. In the second view, it maps the high-level formula Add to the low-level formula $(Add \text{ WITH } \ldots)$. It may not be obvious, but these two views are equivalent. They're equivalent because

(5.3) $[\![Add \text{ WITH } \ldots]\!](\sigma) \;\equiv\; [\![Add]\!](\widehat{f}(\sigma))$

is true for every S-behavior σ. To understand why (5.3) is true, remember that the meaning $[\![F]\!]$ of a formula F is defined in terms of the meanings of its subformulas. Consider the subformula $z' = x + y$ of Add, which appears in its next-state action $NextA$. You should understand why (5.3) is true if you understand why this is true:

(5.4) $[\![(z' = x + y) \text{ WITH } \ldots]\!](\sigma) \;\equiv\; [\![z' = x + y]\!](\widehat{f}(\sigma))$

A proof of (5.4) is given in the Appendix. However, I recommend that you try figuring out by yourself why it's true.

The bidirectional nature of refinement mappings—from low-level behaviors to high-level behaviors and from high-level formulas to low-level formulas—is an example of a general mathematical principle. To explain it, we need the concept of function composition. For any functions g and h, their composition $g \bullet h$ is defined to be the function whose domain is the domain of h such that $g \bullet h(d) \triangleq g(h(d))$ for any value d in the domain of h.[1] (For any d in the domain of h, the expression $g \bullet h(d)$ is meaningful only if $h(d)$ is in the domain of g.)

Let's suppose that the collections of S-behaviors and A-behaviors are sets. For example, we can let them be behaviors that satisfy suitable type invariants. Formula (5.3) can then be written:

(5.5) $[\![Add \text{ WITH } \ldots]\!] \;\equiv\; [\![Add]\!] \bullet \widehat{f}$

In general, suppose L and H are sets and $\psi \in (L \to H)$, so the function ψ maps elements of L to elements of H. For any set V, the function ψ determines a function $\overleftarrow{\psi}$ in $(H \to V) \to (L \to V)$, so $\overleftarrow{\psi}$ maps functions from H to V to functions from L to V. The function $\overleftarrow{\psi}$ is defined by $\overleftarrow{\psi}(g) \triangleq g \bullet \psi$. Pictorially, we have:

negative vspace added

$$\begin{tikzcd} L \arrow[r, "\psi"] & H \\ (L \to V) & (H \to V) \arrow[l, "\overleftarrow{\psi}"'] \end{tikzcd}$$

Refinement mappings are the special case in which L is the set of S-behaviors, H is the set of A-behaviors, V is the set $\{\text{TRUE}, \text{FALSE}\}$ of Booleans, and ψ is $\widehat{f}$.

[1]Mathematicians generally use "$\cdot$" or "$\circ$" to denote function composition, but we have defined those symbols to mean other things.

5.1.3 A Step and Data Refinement

Section 3.5.1 illustrated step refinement by showing that an hour-minute-second clock refines an hour-minute clock. We have just illustrated data refinement by showing that *AddS* implements *Add* under a refinement mapping. We now show an example that involves both step and data refinement.

The example involves an algorithm *AddSeq* that adds numbers the way you probably learned to add them as a child. However, we represent those numbers as sequences of digits in the reverse order, as they are in program *AddS*, with the low-order digit being the first one in the sequence. It sets *sum* equal to $u \oplus v$, where u and v are numbers represented by strings of digits. The algorithm computes *sum* digit by digit, keeping it equal to the right-most digits of the sum computed so far. Each step removes the first (right-most) digit from u and v and appends the next (left-most) digit to *sum*, setting *carry* equal to the value 0 or 1 "carried over" from that sum.

A pseudocode description of the algorithm is in Figure 5.1, but it uses some notation that requires explanation. Recall that $Append(seq, val)$ is defined in Section 2.3.2 to equal $seq \circ \langle val \rangle$, the sequence obtained by appending the value *val* to the end of the sequence *seq* of values. The code assumes that *DigitSeq* is the set $Seq(1\,..\,9) \setminus \{\langle\,\rangle\}$ of nonempty finite sequences of the digits 0 through 9. If u and v are of unequal length, then the number of steps taken by the algorithm you learned in school is usually equal to or one greater than the length of the shorter number. For simplicity, the number of steps taken by *AddSeq* always equals one plus the length of the longer number. To simplify the description of what happens when the algorithm runs out of digits in one of the numbers, it uses the operator *Fix* defined as follows to replace the empty sequence by $\langle 0 \rangle$:

$$Fix(seq) \;\triangleq\; \text{IF } seq = \langle\,\rangle \text{ THEN } \langle 0 \rangle \text{ ELSE } seq$$

The algorithm's **define** statement defines *digit* to equal the indicated expression within that statement. The value of $\lfloor n/10 \rfloor$ is the greatest integer less than or equal to $n/10$. To simplify the invariant, *AddSeq* specifies the initial value of *carry* to equal 0 and ensures that it equals 0 at the end. Since the low-order digit of a two-digit number n is $n \,\%\, 10$ and its high-order digit is $\lfloor n/10 \rfloor$, it should be clear that *AddSeq* describes an algorithm for adding two decimal numbers. (If it's not, execute it by hand on an example.)

The usual way to express correctness of a program that computes a value *sum* and stops is with an invariant asserting that if the program has stopped then *sum* has the correct value. We can't do that with *AddSeq* because the correct value of *sum* is the initial value of $u \oplus v$, and those initial values

```
**variables** $u \in \mathit{DigitSeq},\ v \in \mathit{DigitSeq},\ \mathit{sum} = \langle\,\rangle,\ \mathit{carry} = 0,\ \mathit{pc} = a$ ;
**while** $a$: $(u \neq \langle\,\rangle) \lor (v \neq \langle\,\rangle) \lor (\mathit{carry} \neq 0)$ **do**
  **define** $\mathit{digit} \triangleq \mathit{Fix}(u)(1) + \mathit{Fix}(v)(1) + \mathit{carry}$;
    $\mathit{sum} := \mathit{Append}(\mathit{sum}, \mathit{digit}\,\%\,10)$ ;
    $\mathit{carry} := \lfloor \mathit{digit} / 10 \rfloor$
  **end define** ;
  $u := \mathit{Tail}(\mathit{Fix}(u))$ ;
  $v := \mathit{Tail}(\mathit{Fix}(v))$
**end while** ;
$\mathit{carry} := 0$
```

Figure 5.1: Algorithm *AddSeq*.

have disappeared by the time the program stops. To express correctness, we can add a constant *ans* that equals the initial value of $u \oplus v$. Since stopping means *pc* equals *done* for our pseudocode, correctness means:

(5.6) $\models \mathit{AddSeq} \Rightarrow \Box((\mathit{pc} = \mathit{done}) \Rightarrow (\mathit{ans} = \mathit{sum}))$

The key part of an inductive invariant to prove (5.6) is the assertion that *ans* equals the final value of *sum*. A first approximation to the final value of *sum* is:

$$\mathit{sum} \circ (\langle\,\mathit{carry}\,\rangle \oplus (u \oplus v))$$

We haven't said what $s \oplus t$ means if s or t is the empty sequence, but it's clear that we should define the empty sequence to represent 0. However, a close examination of the algorithm indicates that if u and v both equal $\langle\,\rangle$ and $\mathit{carry} = 0$, then this expression equals a sequence with an extra 0 at the end. The correct assertion that is the key to the inductive invariant is

(5.7) $$\mathit{ans} = \text{IF } (u \circ v = \langle\,\rangle) \land (\mathit{carry} = 0) \text{ THEN } \mathit{sum} \text{ ELSE } \mathit{sum} \circ (\langle\,\mathit{carry}\,\rangle \oplus (u \oplus v))$$

The remainder of the inductive invariant asserts type correctness and that $\mathit{pc} = \mathit{done}$ implies that u, v, and *carry* have their correct final values.

However, the point of this example is not that *AddSeq* implements a particular procedure for adding sequences of digits; it's that it refines the abstract program *Add* that adds two integers in a single step. This is a dramatic example of step refinement, in which a program that can take arbitrarily many non-stuttering steps to finish refines one that always finishes in one non-stuttering step. And we don't have to add the constant *ans* to do it.

Under the refinement mapping, one step in an execution of *AddSeq* must refine a *NextA* step of *Add*; all the other steps must refine stuttering steps of *AddSeq*. The initial values of the variables x and y of *Add* should equal the initial values of $Val(u)$ and $Val(v)$. The initial values of u and v are no longer deducible from the state after *AddSeq* takes it first step. This tells us that the *NextA* step of *Add* must be refined by the first non-stuttering step of *AddSeq*.

An *Add* step changes the value of its variable *done* from FALSE to TRUE. So, the refinement mapping must assign to *done* an expression whose value is changed from FALSE to TRUE by the first non-stuttering step of *AddSeq*. Since further steps of *AddSeq* refine stuttering steps of *Add*, the expression assigned to *done* must remain true for the rest of the execution of *Add*. A suitable expression is $sum \neq \langle\,\rangle$, so we let the refinement mapping include $done \leftarrow sum \neq \langle\,\rangle$.

In the initial state of *AddSeq*, the refinement mapping should assign to x and y the values of u and v. Since *Add* allows x and y to have any values in its final state, it doesn't matter what values the refinement mapping assigns to x and y after the first step of *AddSeq*. However, since later steps must refine stuttering steps of *Add*, the values of x and y must not change. Zero seems like a nice value to let x and y equal when their value no longer matters, so we let the refinement mapping include:

$$\begin{aligned} x &\leftarrow \text{IF } sum \neq \langle\,\rangle \text{ THEN } 0 \text{ ELSE } Val(u)\ , \\ y &\leftarrow \text{IF } sum \neq \langle\,\rangle \text{ THEN } 0 \text{ ELSE } Val(v) \end{aligned}$$

Finally, we must decide what value the refinement mapping assigns to z. If we add to *AddSeq* the constant *ans* that always equals the result the algorithm finally computes, then we can substitute $Val(ans)$ for z. But we don't have to add it because the invariant (5.7) tells us what expression containing only the variables of *AddSeq* always equals *ans*. We could therefore substitute for z the expression obtained by applying *Val* to the right-hand side of equation (5.7). However, there's a simpler expression that we can use. Convince yourself that the following substitution works:

$$z \leftarrow Val(sum) + 10^{Len(sum)} * (carry + Val(u) + Val(v))$$

This completes the refinement mapping. That *AddSeq* implements *Add* under the refinement mapping means that this theorem is true:

$$\begin{array}{l} \models AddSeq \Rightarrow (Add \ \textsc{with} \\ \qquad\qquad done \leftarrow sum \neq \langle\,\rangle \ , \\ \qquad\qquad x \leftarrow \textsc{if} \ \ sum \neq \langle\,\rangle \ \ \textsc{then} \ \ 0 \ \ \textsc{else} \ \ Val(u) \ , \\ \qquad\qquad y \leftarrow \textsc{if} \ \ sum \neq \langle\,\rangle \ \ \textsc{then} \ \ 0 \ \ \textsc{else} \ \ Val(v) \ , \\ \qquad\qquad z \leftarrow \ Val(sum) + 10^{Len(sum)} * (carry + Val(u) + Val(v))) \end{array}$$

5.2 Invariance Under Refinement

If an abstract program T implements an abstract program S under a refinement mapping, and Inv is an invariant of S, then the refinement mapping maps Inv to an invariant of T. The precise statement of this is the following theorem, where "$\ldots$" is any refinement mapping.

Theorem 5.1 $\models T \Rightarrow (S \ \textsc{with} \ldots)$ and $\models S \Rightarrow \Box Inv$ imply $\models T \Rightarrow \Box(Inv \ \textsc{with} \ldots)$.

The proof is simple:

1. $\models S \Rightarrow \Box Inv$ implies $\models (S \Rightarrow \Box Inv) \ \textsc{with} \ldots$.

   Proof: Substitution in a true formula produces a true formula

2. $\models (S \Rightarrow \Box Inv) \ \textsc{with} \ldots$ equals $\models (S \ \textsc{with} \ldots) \Rightarrow \Box(Inv \ \textsc{with} \ldots)$.

   Proof: By definition of what substitution means.

3. Q.E.D.

   Proof: The theorem follows from steps 1 and 2 by propositional logic.

Recall the trick used in Section 3.2.3 to obtain an invariant of *FGSqrs* from an invariant of the coarser-grained algorithm *Sqrs*. We replaced the variable y by the expression yy in the invariant of *Sqrs*. That trick was an application of the theorem, because *FGSqrs* implements *Sqrs* under the refinement mapping $x \leftarrow x, y \leftarrow yy$.

5.3 An Example: The Paxos Algorithm

We've seen step and data refinement for a sequential abstract program. Concurrency adds nothing new. Refinement works exactly the same for concurrent programs. This is illustrated with the Paxos consensus algorithm, an example chosen for the following reasons:

- It's a distributed algorithm. Quite a few researchers used to believe that different techniques are needed to reason about correctness of distributed programs; perhaps some still do. Paxos illustrates that there is no mathematical difference between distributed and non-distributed concurrent programs. In fact, Paxos is obtained as a refinement of a non-distributed algorithm.

- It's a widely used algorithm. If you perform any commercial transaction on the Web, there is a good chance that Paxos or an algorithm inspired by it is being executed by a program running on the computers that perform the transaction.

- It illustrates the importance of abstraction. Thinking scientifically means thinking abstractly. The abstract programs in this example are more abstract than ones most computer scientists and engineers would think of. Learning to think more abstractly is the key to building better complex computer systems.

- The complete TLA$^+$ specifications of the abstract programs, as well as videos of a pair of lectures that explain them, are available on the Web [26]. Therefore, the abstract programs are only sketched here.

The Paxos algorithm was invented (at least) twice, first by Barbara Liskov and Brian Oki [41] and then by me.

5.3.1 The Consensus Problem

One reason for building distributed systems is fault tolerance. Systems implemented by multiple computers are often required to operate normally even if one or more of the computers fail. What a system should do can be described as a single-process abstract program that executes a sequence of commands it receives as inputs. Correct execution of the system by multiple computers requires that all the computers agree on what that sequence of inputs is. This is achieved by having all the computers agree on what the i^{th} input is for every i. Ensuring that all the computers agree on a single input is called *consensus*. A fault-tolerant system repeatedly executes a consensus algorithm to choose a sequence of inputs.

I was inspired to invent the Paxos algorithm because colleagues were building a distributed fault-tolerant system. I realized that the system had to implement consensus, so it should implement a consensus algorithm. However, my colleagues were writing code; they didn't have an algorithm. I never found out how their program implemented consensus. But based on the state of the art of programming at the time, here is what their program might have done.

A process called the *leader*, running on a single computer, receives all input requests and decides what input should be chosen next. A new leader will have to be selected if the initial leader fails, but we'll worry about that later. (Failure of a process usually means failure of the computer executing the process.) For the system to keep running despite the failure of individual computers, a set of processes called *acceptors*, each running on a different computer, have to know what value was chosen. Moreover, only a subset of the acceptors should have to be working (that is, not failed) for an input to be chosen. If an input v is chosen by a leader and a set of acceptors, and the leader and those acceptors fail, then a different leader and a different set of acceptors must not choose an input different from v. The obvious way to ensure that is to require a majority of the acceptors to agree upon the input v in order for that input to be chosen. Any two majorities have at least one acceptor in common, and that acceptor will know that it agreed to the choice of v.

This reasoning leads to the following algorithm: The leader decides what input v should be chosen. It sends a message to the acceptors saying that they should agree to the choice of v. Any working acceptor that receives the message replies to the leader with a message saying "v is OK". When the leader receives such an OK message from a majority of acceptors, it sends a message to all the acceptors telling them that v has been chosen.

This algorithm works fine, and the system keeps choosing a sequence of inputs, until the leader fails. At that point, a new leader is selected. The new leader sends a message to all the acceptors asking them what they've done. In particular, the new leader finds out from the acceptors if inputs were chosen that it was unaware of. It also finds out if the previous leader had begun trying to choose an input but failed before the input was chosen. If it had, then the new leader completes the choice of that input. When the new leader has received this information from a majority of acceptors, it can complete any uncompleted choices of an input and begin choosing new inputs. Let's call this algorithm the naive consensus algorithm.

There's one problem with the naive algorithm: How is the new leader chosen? Choosing a single leader is just as hard as choosing a single input. The naive consensus algorithm thus assumes the existence of a consensus algorithm. However, because leader failures should be rare, choosing a leader does not have to be done efficiently. So, programmers would probably have approached the problem of choosing a leader the way they approached most programming problems. They would have found a plausible solution and then debugged it. Debugging usually means thinking of all the things that could go wrong and adding code to handle them.

Let's pause and look at the science of consensus. Before Paxos, there were consensus algorithms that worked no matter what a failed process could do [45]. However, they were *synchronous* algorithms, meaning that they assumed known bounds on the time required for messages sent by one process to be received and acted upon by another process. They were not practical for the loosely coupled computers that had become the norm by the 1980s. Although asynchronous algorithms were required, they had to solve a simpler problem because sufficiently reliable systems could be based on the assumption that a process failed by stopping and could not perform incorrect actions. However, the FLP theorem, named after Michael Fischer, Nancy Lynch, and Michael Paterson who discovered and proved it, states that no asynchronous algorithm can implement consensus if even a single process can fail in this benign way [13]. More precisely, any algorithm that ensures the safety property that two processes never choose different values must allow behaviors that violate the liveness property that requires a value eventually to be chosen if enough processes are working and can communicate with one another. Asynchronous algorithms that ensure liveness must allow behaviors in which processes disagree about what input is chosen.

The leader-selection code programmers would have written therefore had to allow either behaviors in which two processes thought they were the leader, probably with serious consequences, or else behaviors in which no leader is selected, causing the system to stop choosing values. With a properly designed algorithm, the probability of never choosing the leader is zero, and a leader will be chosen fairly quickly if enough of the system is working properly. The system my colleagues built ran for several years with about 60 single-user computers, and I don't think their consensus code caused any system error or noticeable stalling. There is no way to know if it had errors that would have appeared in today's systems with thousands of computers and many thousands of users.

5.3.2 The Paxos Consensus Algorithm

We develop the Paxos consensus algorithm as a series of three abstract programs: a trivial specification of the problem the algorithm solves, which is refined by a non-distributed multiprocess algorithm, which is refined by the Paxos algorithm. I believe that this description—in particular, its view of Paxos as a refinement of the non-distributed algorithm—mirrors how I actually found the algorithm.

Only the safety properties of these abstract programs are described. In most applications, violation of safety in a consensus algorithm can be quite serious—for example, causing money deposited to a client's bank account to disappear. We will see later how the algorithm can be implemented to almost always achieve liveness while never violating safety. As mentioned above, the abstract programs are just sketched; complete descriptions are available on the Web [26].

5.3.2.1 The Specification of Consensus

Instead of talking about inputs, we define consensus as choosing an element of some set *Value* of values. Most correctness proofs of consensus algorithms prove only that they satisfy the invariance property that two processes never choose different values. A consensus algorithm must also not allow a value to be unchosen and a different value then chosen. Proving the invariance property is usually sufficient because it's obvious that the algorithm doesn't allow a value to be unchosen. But to illustrate refinement, we write a high-level abstract program that rules out such a possibility.

There are a number of reasonable ways to describe consensus as an abstract program, and it makes little difference which one is used. Perhaps the most obvious way is with a multiprocess abstract program in which each process independently learns what value is chosen. The next-state action would allow a process p that has not learned a value to learn one, with the constraint that if any process has learned that the value v was chosen, then p must also learn that v was chosen.

We take a different approach and let the abstract program describe only the choosing of a value, without mentioning processes that learn the chosen value. This abstract program has a single variable *chosen* that represents the set of values that have been chosen. (In any behavior allowed by the program, that set always has at most one value.) The initial predicate is $chosen = \{\}$, and the next-state action is:

$$(chosen = \{\}) \ \wedge \ (\exists\, v \in Value \ : \ chosen' = \{v\})$$

As explained above, there is no fairness condition.

5.3.2.2 The Voting Algorithm

In the naive algorithm, leader and acceptor processes communicate by sending messages. It's natural to think about a consensus algorithm in terms of messages being sent. However, remember that we reason about an abstract program in terms of its state, so we should be thinking about states, not about sending messages. And the important part of the state is the state of the acceptors. So, we refine the Consensus program with an abstract program called the Voting algorithm whose state is just the state of the acceptors. This is not just a nice way to describe the Paxos algorithm. I believe it describes how I was actually thinking when I discovered Paxos.

In good programming, we begin by abstracting away lower-level details and getting the high-level design right. There's a kind of bad programming that sounds similar: We begin writing something that handles the normal behavior, and we then modify it to handle non-normal situations. That's the way the naive consensus algorithm was described, and it's a recipe for creating incorrect programs—both abstract and concrete ones. We should start thinking about the general case, not the normal case.

The general state of acceptors in the naive algorithm is one that is reached after a number of leaders have begun trying to get a value chosen, and some of them may have succeeded. When a leader tries to get a particular value chosen, we say that the leader has begun a *ballot*. When an acceptor has sent an OK message for a value v in that ballot, we say that the acceptor has *voted* for v in that ballot. The algorithm will assign a unique natural number to each ballot.[2] The state of the Voting algorithm records all the votes that each acceptor has cast. This is described by a variable *votes* whose value is a function that assigns to each acceptor a a set $votes(a)$ of pairs $\langle b, v \rangle$ where $b \in \mathbb{N}$ and $v \in Value$. The pair $\langle b, v \rangle$ in $votes(a)$ means that a has voted for v in ballot number b.

Choosing a leader is the weak point in the naive algorithm. The Voting algorithm abstracts away the leaders. A leader serves two functions. The first is to ensure that in any ballot, acceptors can cast votes only for the value proposed by the leader. The Voting algorithm's next-state action takes care of that by not letting an acceptor cast a vote for a value v in ballot b if a vote has already been cast in ballot b for a different value. The second function of the leader is to learn that a value has been chosen, which it does when it has received enough OK messages. The Voting algorithm does away with that function by declaring that the value has been chosen when the requisite number of OK messages have been sent—that is, when there are enough votes cast for the value in the ballot. More precisely, we define $ChosenAt(b, v)$ to be true iff a majority of acceptors has voted for v in

[2]Don't confuse different ballots with the different instances of the consensus algorithm being executed. Execution of an instance of the consensus algorithm can consist of multiple ballots.

ballot b. The Voting algorithm implements the Consensus abstract program under the refinement mapping

(5.8) $chosen \;\leftarrow\; \{v \in Value \,:\, \exists\, b \in \mathbb{N} \,:\, Chosen(b, v)\}$

In addition to *votes*, the algorithm has one other variable *maxBal* whose value is a function that assigns to each acceptor a a number $maxBal(a)$. The significance of this number is that a will never in the future cast a vote in any ballot numbered less than $maxBal(a)$. The value of $maxBal(a)$ is initially 0 and is never decreased. The algorithm can increase $maxBal(a)$ at any time.

It may seem strange that the state does not contain any information about what processes have failed. We are assuming that a failed process does nothing. Since we are describing only safety, a process is never required to do anything, so there is no need to tell it to do nothing. A failed process that has been repaired can differ from a process that hasn't failed because it may have forgotten its prior state when it resumes running. A useful property of a consensus algorithm is that, even if all processes fail, the algorithm can resume its normal operation when enough processes are repaired. To achieve this, we require that a process maintains its state in stable storage, so it is restored when a failed process restarts. A process failing and restarting is then no different from a process simply pausing.

The heart of the Voting algorithm is a state expression $SafeAt(b, v)$ that is true iff $ChosenAt(c, w)$ is false and will remain false forever for any $c < b$ and $w \neq v$. That it will remain false forever can be deduced from the current state, because the next-state action implies both that a process a will not cast a vote in ballot c when $c < maxBal(a)$ and that $maxBal(a)$ can never decrease. The key invariant maintained by the algorithm is

(5.9) $\forall\, a \in Acceptor, b \in \mathbb{N}, v \in Value \,:$
$$(\langle\, b, v\,\rangle \in votes(a)) \;\Rightarrow\; SafetAt(b, v)$$

where *Acceptor* is the set of acceptors. The next-state action allows a process a to perform either of two actions:

- Increase $maxBal(a)$. This action is always enabled.
- Vote for a value v in a ballot numbered b. As already explained, this action is enabled only if no process has voted for a value other than v in ballot b and $b \geq maxBal(a)$. An additional enabling condition is required to maintain the invariance of (5.9).

I have given you all the information you need to figure out the definition of *SafeAt*(b, v) and the enabling condition on acceptors needed to maintain the invariance of (5.9). Can you do it? Few people can. I was able to only because I had simplified the problem to finding an abstract program whose only processes are the acceptors and whose state consists only of the set of votes cast and the value of *maxBal*. I had abstracted away leaders, messages, and failures.

The Voting algorithm requires an acceptor to know the current state of other acceptors to decide what vote it can cast. How can this lead to a distributed consensus algorithm? I abstracted away leaders and messages; I didn't ignore them. I knew that an acceptor didn't have to directly observe the state of other acceptors to know that they hadn't voted for some value other than v in a ballot. The acceptor could know that because of a message it received from a leader. I also knew that it could deduce that the other enabling conditions were satisfied from messages it received. Abstracting away leaders and messages enabled me to concentrate on the core problem of achieving consensus. The solution to that problem told me what the leaders should do and what messages needed to be sent.

5.3.2.3 The Paxos Abstract Program

The Voting algorithm told me what messages needed to be sent. But I had to decide how to represent message passing in an abstract program. Languages expressly designed for describing distributed algorithms usually don't require us to make that decision because they provide built-in message-passing primitives. However, different distributed algorithms and distributed systems have different requirements for message passing. They may or may not tolerate lost messages; they may or may not require messages to be delivered in the order they are sent; they may or may not require that the same message not be received twice; and so on. Our abstract programs require that we choose how to represent message passing, but they make it easy to represent any form of message passing we want.

I have found that most computer scientists and engineers are constrained by thinking in terms of how messages are transmitted in actual systems. They think of messages being sent on communication channels between processes. Few of them would come up with the simple representation of message passing I used in the Paxos abstract program—a representation that is obvious if one thinks mathematically.

Paxos doesn't require that messages be delivered in the order in which they are sent, so there is no need for message channels. The receiver of a message can be inferred from the message, so we can just have a set of messages. Paxos tolerates the same message being received multiple times by a process, so there is no need to remove a message when it is received. This means that if the same message is sent to multiple recipients, there is no need for multiple copies of the message. There is also no need for a separate action of receiving a message. An action that should be taken upon receipt of a message simply has the existence of that message in the set of sent messages as an enabling condition. Paxos tolerates message loss. But since we are describing safety, there's no difference between a lost message and a message that is sent but never received. So, there is no need ever to remove messages that have been sent.

We can therefore represent message passing with a variable *msgs* whose value is the set of all messages that have been sent. A message is sent by adding it to the set *msgs*. The presence of a message in *msgs* enables an action that should be triggered by the receipt of the message. The algorithm has a variable *maxBal* that implements the variable of the same name in the Voting algorithm. It also has two other variables *MaxVBal* and *MaxVal* whose values are functions with domain the set of acceptors. They are explained below.

The Paxos consensus algorithm can be viewed as a multiprocess algorithm containing two sets of processes: the acceptors that implement the acceptors of the Voting algorithm, and an infinite set of processes, one for each natural number, where process number b is the leader of ballot number b. More precisely, the ballot b leader orchestrates the voting by the acceptors in ballot b of the Voting algorithm.

The next-state action of the algorithm could be (but isn't literally) written in the form $\exists\, b \in \mathbb{N} : BA(b)$ where $BA(b)$ describes how ballot b is performed. The ballot consists of two phases. In phase 1, the ballot b leader sends a message to the acceptors containing only the ballot number b. An acceptor a ignores the message unless $b > maxBal(a)$, in which case it sets $maxBal(a)$ to b and replies with a message containing a, b, $MaxVBal(a)$, and $MaxBal(a)$. When the ballot b leader receives those messages from a majority of the acceptors, it can pick a value v to be chosen, where v is either a value picked by the leader of a lower-numbered ballot or an arbitrary value. The complete algorithm describes how it picks v. Phase 2 begins with the leader sending a message to the acceptors asking them to vote for v in ballot b. An acceptor a ignores the message unless $b \geq maxBal(a)$, in which case a sets $maxBal(a)$ to b and replies with a message saying that it has voted for v in ballot b.

The Paxos algorithm implements the Voting algorithm under a refinement mapping in which the variable *votes* of Voting is implemented by the expression defined in the obvious way from the set of votes reported by acceptors' phase 2 messages in *msgs*, and in which the variable *maxBal* of Voting is implemented by the variable of the same name in the Paxos abstract program.

The values of *maxVBal* and *maxVal* can be described as functions of the value of *votes*. For any acceptor a, the pair $\langle maxVBal(a), maxVal(a) \rangle$ equals the pair $\langle b, v \rangle$ in the set $votes(a)$ with the largest value of b. (Initially, when $votes(a)$ is the empty set, it equals $\langle -1, None \rangle$ for some special value *None*.) Making *maxVBal* and *maxVal* variables rather than state expressions makes it clear that they are the only information about what messages have been sent that needs to be part of the acceptors' states.

5.3.3 Implementing Paxos

An implementation of the Paxos consensus algorithm would add a third phase to each ballot in which the leader sends a message announcing that a value v had been chosen after it receives phase 2 messages telling it that a majority of acceptors had voted for v in the ballot. With that addition, a ballot of the Paxos algorithm looks like what the naive algorithm does when a new leader has been selected. Phase 1 of a Paxos ballot corresponds to the new leader finding out if it needs to complete the choosing of a value proposed by the failed leader. Phase 2 corresponds to the leader completing the choosing of a previously proposed value or choosing a new value.

In Paxos, a leader performs phase 1 in every instance of the consensus algorithm, while in the naive algorithm it performs the corresponding actions only once when it is selected. This makes Paxos seem much less efficient, since leaders are infrequently replaced. However, the value to be chosen isn't selected until phase 2. This means that phase 1 can be executed simultaneously for a ballot numbered b in all instances of the consensus algorithm the first time ballot b is executed for any instance. A single message can serve as the leader's phase 1 message for all instances. A single message can also contain the phase 1 responses of a particular acceptor for all the instances, since there is only information to be transmitted for consensus instances that have begun but not yet chosen a value. Thus Paxos uses the same number of messages to choose a value as the naive algorithm.

The Paxos consensus algorithm is an efficient algorithm that has been proved to satisfy the safety requirement of consensus. We still have to see how to get it to satisfy the liveness requirement of actually choosing a value. To solve a problem, we need to understand it, and it's easy to understand what prevents Paxos from choosing a value. An acceptor a participating in ballot number b sets $maxBal(a)$ to b, preventing it from responding to any message from the leader of any ballot numbered less than b. Even in the absence of failures or message loss, no value will ever be chosen if higher and higher numbered ballots are begun before any ballot chooses a value.

Conversely, if ballot number b is started and no higher-numbered ballot is begun, and if the ballot b leader and a majority of acceptors are working, then liveness assumptions that require working processes eventually to perform enabled actions (which implicitly assume that messages sent are eventually delivered) imply that a value is eventually chosen. This observation can be stated mathematically as a temporal logic formula and proved. However, it is so obviously true that, to my knowledge, no one has ever bothered doing it.

How do we assure that a ballot numbered b is started and no higher-numbered ballots are? Paxos uses an infinite number of leader processes—one for each ballot number. Those infinitely many processes are executed by a finite number of computers, with each ballot number pre-assigned to a single computer that executes the leader of the corresponding ballot. A single computer, called the coordinator, is selected to be the only one that executes leader processes, and it is easy to add messages that allow it to find a ballot number higher than the values of $mbal(a)$ for a majority of acceptors a.

Like the naive algorithm, Paxos depends on selecting a single coordinator. However, the naive algorithm can fail to maintain its safety requirement if two different computers believe they are the coordinator. If that happens with Paxos, safety is preserved; the algorithm just fails to make progress. An algorithm for choosing a coordinator in Paxos needs to work only most of the time, a much easier problem to solve. One solution uses a synchronous algorithm that implements consensus assuming known bounds on the times needed to transmit and process messages. That algorithm chooses the coordinator assuming values for those bounds that will be satisfied most of the time.

5.4 Proving Refinement

This section sketches how to prove that one abstract program refines another. We use as an example the proof that the One-Bit mutual exclusion algorithm *OB* of Section 4.2.5.2 refines program *LM* of Figure 4.6, assuming a weakly fair semaphore.

The proof uses identifiers from the definition of OB and ones from the definition of LM. To avoid confusion, we indicate to which program an identifier belongs with a subscript. We defined OB to equal:

$$
\begin{aligned}
&\mathit{Init} \land \Box[\mathit{Next}]_v \land \mathit{Fair} \\
&\textbf{where } \mathit{Next} \triangleq \exists\, p \in \{0,1\} : \mathit{PNext}(p) \\
&\phantom{\textbf{where }} \mathit{Fair} \triangleq \forall\, p \in \{0,1\} : \mathrm{WF}_v(\mathit{PNext}(p))
\end{aligned}
$$

We now add subscripts to that definition, so OB equals:

$$\mathit{Init}_{OB} \land \Box[\mathit{Next}_{OB}]_{v_{OB}} \land \mathit{Fair}_{OB}$$

We assume LM has the same definition, except with the subscripts LM. We sometimes use subscripts even when they aren't necessary—for example writing x_{OB} even though LM has no variable named x.

We define OBSafe and OBFair as before, so OB equals $\mathit{OBSafe} \land \mathit{OBFair}$. We define LMSafe and LMFair similarly. As expected, safety and liveness are proved separately. We first show that OBSafe refines LMSafe. (By machine closure of $\langle \mathit{OBSafe}, \mathit{OBFair} \rangle$ and Theorems 4.2 and 4.4, OBFair isn't needed to prove that OB refines LMSafe.) We then show that OB implies LMFair. However, first we must define the refinement mapping under which OB implements LM.

5.4.1 The Refinement Mapping

To define the refinement mapping, it's helpful to think of a single behavior in which the variables x_{OB} and pc_{OB} describe a behavior of program OB and the variables sem_{LM} and pc_{LM} describe the corresponding behavior of LM that is implemented by OB under the refinement mapping. To determine what the refinement mapping should be, for each possible step in such a behavior that changes the values of the variables of LM, we decide how that step should change the values of OB.

For example, if a step of the behavior describes the execution of statement cs by a process p of LM, then it should describe the execution of cs by process p of OB. Thus, when the value of $pc_{LM}(p)$ changes from cs to $exit$, the value of $pc_{OB}(p)$ should also change from cs to $exit$. Reasoning in this way, we see that the values of $pc_{LM}(p)$ and $pc_{OB}(p)$ should be equal except that when $pc_{LM}(p)$ equals $wait$, the value of $pc_{OB}(p)$ can be $wait$, $w2$, $w3$, or $w4$. This tells us that the refinement mapping must substitute for pc_{LM} the value $pcBar_{OB}$ defined by:

$$
\mathit{pcBar}_{OB} \triangleq p \in \{0,1\} \mapsto \textsc{if } pc_{OB}(p) \in \{w2, w3, w4\} \textsc{ then } \mathit{wait} \textsc{ else } pc_{OB}(p)
$$

In a behavior satisfying LM, the value of sem_{LM} can be deduced from the value of pc_{LM}. In particular, sem_{LM} equals 0 iff $pc_{LM}(p)$ equals pc or $exit$ for one of the processes p. From the definition of $pcBar_{OB}$, this means that the refinement mapping must substitute for sem_{LM} the value $semBar_{OB}$ defined by:

$$semBar_{OB} \triangleq \text{IF } (\exists\, p \in \{0,1\} : pc_{OB}(p) \in \{cs, exit\}) \text{ THEN } 0 \text{ ELSE } 1$$

That OB refines LM under this refinement mapping means:

$$(5.10)\quad \models OB \;\Rightarrow\; (LM \text{ WITH } pc_{LM} \leftarrow pcBar_{OB},\ sem_{LM} \leftarrow semBar_{OB})$$

We'll be dealing with a lot of formulas obtained from a formula F_{LM} by making the substitutions defined by the refinement mapping for the variables of LM. To keep from having lots of WITHs, we use this abbreviation, for any formula F_{LM}:

$$\overline{F_{LM}} \;\triangleq\; (F_{LM} \text{ WITH } pc_{LM} \leftarrow pcBar_{OB},\ sem_{LM} \leftarrow semBar_{OB})$$

Thus, (5.10) can be written $\models OB \Rightarrow \overline{LM}$. Also, $\overline{pc_{LM}}$ equals $pcBar_{OB}$ and $\overline{sem_{LM}}$ equals $semBar_{OB}$ (which explains the suffix Bar in the names $pcBar$ and $semBar$).

5.4.2 Refinement of Safety

We now sketch the proof that $OBSafe$ refines $LMSafe$, which means the proof of

$$(5.11)\quad \models OBSafe \Rightarrow \overline{LMSafe}$$

By the definitions of these formulas, this requires proving:

$$(5.12)\quad \begin{array}{ll} \text{(a)} & \models Init_{OB} \Rightarrow \overline{Init_{LM}} \\ \text{(b)} & \models Init_{OB} \wedge \Box[Next_{OB}]_{v_{OB}} \;\Rightarrow\; \Box\overline{[Next_{LM}]_{v_{LM}}} \end{array}$$

The proof of (a) is simple. To prove (b), we use the invariant Inv_{OB} of OB, which is defined by (4.6), where $TypeOK$ is defined by (4.17). That is, we assume:

$$(5.13)\quad \models OBSafe \Rightarrow \Box Inv_{OB}$$

To prove (5.12b), it suffices to prove

$$(5.14)\quad \models Inv_{OB} \wedge [Next_{OB}]_{v_{OB}} \Rightarrow \overline{[Next_{LM}]_{v_{LM}}}$$

By definition of $[\ldots]_v$, we can prove (5.14) by proving:

$$\text{(5.15)}\quad \begin{array}{ll} \text{(a)} & \models Inv_{OB} \land Next_{OB} \;\Rightarrow\; \overline{Next_{LM}} \lor (\overline{v_{LM}}' = \overline{v_{LM}}) \\ \text{(b)} & \models (v_{OB}' = v_{OB}) \;\Rightarrow\; (\overline{v_{LM}}' = \overline{v_{LM}}) \end{array}$$

Part (b) is trivial, since $\overline{v_{LM}}$ is defined in terms of the variables of v_{OB}. For part (a), propositional logic tells us that we prove $F \land (G_1 \lor \ldots \lor G_n) \Rightarrow H$ by proving $F \land G_i \Rightarrow H$ for each i. So, we decompose the proof of part (a) by writing $Next_{OB}$ as the disjunction of subactions.

We use the notation introduced in Section 4.2.6 of naming the action described by a labeled statement with the capitalized label. For example, $Cs_{OB}(p)$ is the action described by statement *cs* of process p of program OB. We decompose the proof of (5.15a) into proving:

$$\text{(5.16)}\quad \models Inv_{OB} \land Lbl_{OB}(p) \;\Rightarrow\; \overline{Next_{LM}} \lor (\overline{v_{LM}}' = \overline{v_{LM}})$$

for each label *lbl* in Figure 4.3 and p in $\{0, 1\}$.

Condition (5.16) asserts that a step of OB described by the statement labeled *lbl* implements a step of LM under the refinement mapping. We defined the refinement mapping to make that true, so we should be able to prove this assertion. We prove it by showing that each action $Lbl_{OB}(p)$ implements some particular subaction of $Next_{LM}$. In particular, we prove the following seven assertions R1–R7. Three of them assert that actions of OB imply actions of the form $\overline{\langle A_{LM}\rangle_{v_{LM}}}$. For proving that OB implies $\overline{LMSafe}$, we need only the weaker assertions obtained by replacing such an action by $\overline{A_{LM}}$. However, we will need the stronger assertions later for proving that OB implies $\overline{LMLive}$.

R1. $\models Inv_{OB} \land Ncs_{OB}(p) \;\Rightarrow\; \overline{Ncs_{LM}(p)}$

R2. $\models Inv_{OB} \land Wait_{OB}(p) \Rightarrow (\overline{v_{LM}}' = \overline{v_{LM}})$

R3. $\models Inv_{OB} \land W2_{OB}(p) \;\Rightarrow$
$$\begin{array}{l} \textsc{if}\ p = 0\ \textsc{then}\ \overline{\langle Wait_{LM}(0)\rangle_{v_{LM}}} \\ \qquad\qquad \textsc{else}\ \ \textsc{if}\ x_{OB}(0)\ \textsc{then}\ \overline{v_{LM}}' = \overline{v_{LM}} \\ \qquad\qquad\qquad\qquad\qquad\quad \textsc{else}\ \ \overline{\langle Wait_{LM}(1)\rangle_{v_{LM}}} \end{array}$$

R4. $\models Inv_{OB} \land W3_{OB}(p) \;\Rightarrow\; (\overline{v_{LM}}' = \overline{v_{LM}})$

R5. $\models Inv_{OB} \land W4_{OB}(p) \;\Rightarrow\; (\overline{v_{LM}}' = \overline{v_{LM}})$

R6. $\models Inv_{OB} \land Cs_{OB}(p) \;\Rightarrow\; \overline{\langle Cs_{LM}(p)\rangle_{v_{LM}}}$

R7. $\models Inv_{OB} \land Exit_{OB}(p) \;\Rightarrow\; \overline{\langle Exit_{LM}(p)\rangle_{v_{LM}}}$

$$
\begin{array}{ll}
W2_{OB}(0) \;\equiv & Wait_{LM}(0) \;\equiv \\
\quad \land\ pc_{OB}(0) = w2 & \quad \land\ pc_{LM}(0) = wait \\
\quad \land\ \neg x_{OB}(1) & \quad \land\ sem_{LM} = 1 \\
\quad \land\ pc_{OB}{}' = (pc_{OB} \text{ EXCEPT } 0 \mapsto cs) & \quad \land\ pc_{LM}{}' = (pc_{LM} \text{ EXCEPT } 0 \mapsto cs) \\
\quad \land\ x_{OB}{}' = x_{OB} & \quad \land\ sem_{LM}{}' = 0
\end{array}
$$

$$
\begin{array}{lll}
Inv_{OB} & \triangleq & \land\ TypeOK_{OB} \\
 & & \land\ \forall\, p \in \{0,1\} : \begin{array}[t]{l} \land\ (pc_{OB}(p) \in \{w2, cs\}) \Rightarrow x_{OB}(p) \\ \land\ (pc_{OB}(p) = cs) \Rightarrow (pc_{OB}(1-p) \neq cs) \end{array}
\end{array}
$$

$$
\begin{array}{lll}
TypeOK_{OB} & \triangleq & \land\ x_{OB} \in (\{0,1\} \to \{\text{TRUE}, \text{FALSE}\}) \\
 & & \land\ pc_{OB} \in (\{0,1\} \to \{ncs, wait, w2, w3, w4, cs, exit\}) \\
 & & \land\ pc_{OB}(0) \notin \{w3, w4\}
\end{array}
$$

$$
pcBar_{OB} \;\triangleq\; p \in \{0,1\} \mapsto \text{IF } pc_{OB}(p) \in \{w2, w3, w4\} \text{ THEN } wait \text{ ELSE } pc_{OB}(p)
$$

$$
semBar_{OB} \;\triangleq\; \text{IF } \exists\, p \in \{0,1\} : pc_{OB}(p) \in \{cs, exit\} \text{ THEN } 0 \text{ ELSE } 1
$$

$$
\overline{pc_{LM}} \;=\; pcBar_{OB} \qquad \overline{sem_{LM}} \;=\; semBar_{OB}
$$

Figure 5.2: Definitions used in the proofs.

Assertion R3 is equivalent to these three assertions:

R3a. $\models Inv_{OB} \land W2_{OB}(0) \Rightarrow \overline{\langle Wait_{LM}(0) \rangle_{v_{LM}}}$

R3b. $\models Inv_{OB} \land W2_{OB}(1) \land x_{OB}(0) \Rightarrow (\overline{v_{LM}}{}' = \overline{v_{LM}})$

R3c. $\models Inv_{OB} \land W2_{OB}(1) \land \neg x_{OB}(0) \Rightarrow \overline{\langle Wait_{LM}(1) \rangle_{v_{LM}}}$

All these assertions are proved by expanding the definitions of the actions and of the refinement mapping. To see how this works, we consider R3a. We haven't written the definitions of the actions corresponding to the pseudocode statements of algorithms OB and LM. The definitions of $W2_{OB}(0)$ and $Wait_{LM}(0)$ as well as the other relevant definitions are in Figure 5.2. Here is the proof of R3a.

1. SUFFICES: ASSUME: $Inv_{OB} \land W2_{OB}(0)$
   PROVE: $\overline{\langle Wait_{LM}(0) \rangle_{v_{LM}}}$

   PROOF: Obvious.

2. $(\overline{pc_{LM}}(0) = wait) \land (\overline{pc_{LM}}{}' = (\overline{pc_{LM}} \text{ EXCEPT } 0 \mapsto cs))$

Proof: By the step 1 assumption and the definitions of $W2_{OB}(0)$ and $pcBar_{OB}(0)$, since $\overline{pc_{LM}}$ equals $pcBar_{OB}$.

3. $(\overline{sem_{LM}} = 1) \land (\overline{sem_{LM}}' = 0)$

   Proof: $W2_{OB}(0)$ implies $(pc_{OB}(0) = w2) \land \neg x_{OB}(1)$, and Inv_{OB} and $\neg x_{OB}(1)$ imply $pc_{OB}(1) \notin \{cs, exit\}$. Hence, $semBar_{OB} = 1$, so $\overline{sem_{LM}} = 1$. The definition of $W2_{OB}(0)$ and Inv_{OB} (which implies pc_{OB} is a function with domain $\{0, 1\}$) imply $pc_{OB}'(0) = cs$. Hence $semBar_{OB}' = 0$, so $\overline{sem_{LM}}' = 0$.

4. Q.E.D.

   Proof: Steps 2 and 3 and the definition of $Wait_{LM}(0)$ imply $\overline{Wait_{LM}}(0)$. Step 3 implies $\overline{sem_{LM}}' \neq \overline{sem_{LM}}$ which implies $\overline{v_{LM}'} \neq \overline{v_{LM}}$, proving the goal $\overline{\langle\, Wait_{LM}(0) \rangle_{v_{LM}}}$ introduced by step 1.

How we decomposed the proof that *OBSafe* refines *OBLive* into proving R1–R7 was determined by the structure of $Next_{OB}$ as a disjunction of seven subactions and knowing which disjuncts of $\overline{Next_{LM}}$ each of those subactions implements, which followed directly from the definition of the refinement mapping. The decomposition of R3 into R3a–R3c followed from the structure of R3. As illustrated by the proof of R3a, the proof of each of the resulting nine formulas is reduced to ordinary mathematical reasoning by expanding the appropriate definitions. The only place where not understanding the algorithms could result in an error is in the definition of the invariant Inv_{OB} or of the refinement mapping. Catching such an error requires only careful reasoning about simple set theory and a tiny bit of arithmetic, using elementary logic. Someday, computers should be very good at such reasoning.

5.4.3 Refinement of Fairness

This section shows how to prove that a program refines the fairness property of another program by sketching the proof of one example: *OB* implies $\overline{LMFair}$. Define

$$OBB \;\triangleq\; \Box(Inv_{OB} \land \overline{Inv_{LM}}) \;\land\; \Box[Next_{OB}]_{v_{OB}} \;\land\; OBFair$$

where Inv_{OB} is the invariant satisfied by OB defined by (4.6) and (4.17), and Inv_{LM} is an invariant of *LM*. For our example, we just require that Inv_{LM} implies type correctness of *LM*. Formula *OBB* is a $\Box$ formula that is implied by *OB*. (We have proved that *OB* implies $\overline{LMSafe}$, which implies that $\overline{Inv_{LM}}$ is an invariant of *OB*.) We prove $\models OBB \Rightarrow \overline{LMFair}$.

A_{LM}	Q_{OB}	B_{OB}	P_{OB}
$Wait_{LM}(0)$	$\land\ pc_{OB}(0) \in \{wait, w2\}$ $\land\ pc_{OB}(1) \notin \{cs, exit\}$	$W2_{OB}(0)$	$\land\ pc_{OB}(0) = w2$ $\land\ \neg\, x(1)$
$Wait_{LM}(1)$	$\land\ pc_{OB}(1) \in$ $\{wait, w2, w3, w4\}$ $\land\ pc_{OB}(0) \notin \{cs, exit\}$	$W2_{OB}(1)$	$\land\ pc_{OB}(1) = w2$ $\land\ \neg\, x(0)$
$CS_{LM}(p)$	$pc_{OB}(p) = cs$	$CS_{OB}(p)$	$pc_{OB}(p) = cs$
$Exit_{LM}(p)$	$pc_{OB}(p) = exit$	$Exit_{OB}(p)$	$pc_{OB}(p) = cs$

Figure 5.3: Formulas B_{OB}, P_{OB}, and Q_{OB} for the actions A_{LM}, with $p \in \{0, 1\}$.

We use Theorem 4.7 to write *LMFair* as the conjunction of weak fairness of $Wait_{LM}(p)$, $CS_{LM}(p)$, and $Exit_{LM}(p)$, for $p \in \{0, 1\}$. So, we have to prove $\models OBB \Rightarrow \overline{\mathrm{WF}_{v_{LM}}(A_{LM})}$ for A_{LM} equal to each of those six actions. By (4.15), we can do this by proving:

$$(5.17)\quad \models OBB \;\Rightarrow\; (\Box\overline{\mathbb{E}\langle A_{LM}\rangle_{v_{LM}}} \leadsto \langle\overline{A_{LM}}\rangle_{\overline{v_{LM}}})$$

We prove (5.17) by finding an action B_{OB} and state predicates P_{OB} and Q_{OB} satisfying the following conditions:

A1. 1. $\models Inv_{OB} \land \overline{Inv_{LM}} \land \overline{\mathbb{E}\langle A_{LM}\rangle_{v_{LM}}} \;\Rightarrow\; Q_{OB}$

2. $\models OBB \;\Rightarrow\; (\Box Q_{OB} \leadsto \Box P_{OB})$

A2. 1. $\models Inv_{OB} \land \overline{Inv_{LM}} \land P_{OB} \;\Rightarrow\; \mathbb{E}\langle B_{OB}\rangle_{v_{OB}}$

2. $\models OBB \;\Rightarrow\; \mathrm{WF}_{v_{OB}}(B_{OB})$

A3. $\models Inv_{OB} \land \overline{Inv_{LM}} \land P_{OB} \land \langle B_{OB}\rangle_{v_{OB}} \;\Rightarrow\; \langle\overline{A_{LM}}\rangle_{\overline{v_{LM}}}$

To show that these conditions imply (5.17), we have to show that they imply that in any behavior σ satisfying *OBB*, if $\Box\overline{\mathbb{E}\langle A_{LM}\rangle_{v_{LM}}}$ is true of σ^{+m}, then $\sigma(n) \to \sigma(n+1)$ is an $\langle\overline{A_{LM}}\rangle_{\overline{v_{LM}}}$ step for some $n \geq m$. Condition A1.1 implies $\Box Q_{OB}$ is true of σ^{+m}, which by A1.2 implies $\Box P_{OB}$ is true of σ^{+q} for some $q \geq m$. By the definition of WF, conditions A2 imply $\sigma(n) \to \sigma(n+1)$ is a $\langle B_{OB}\rangle_{v_{OB}}$ step for some $n \geq q$, and A3 implies that $\langle B_{OB}\rangle_{v_{OB}}$ step is an $\langle\overline{A_{LM}}\rangle_{\overline{v_{LM}}}$ step.

The formulas B_{OB}, P_{OB}, and Q_{OB} used for the six actions A_{LM} are shown in Figure 5.3. Condition A2.1 for the actions A_{LM} follows easily from the definitions of B_{OB} and P_{OB}. To show that A2.2 is satisfied, we apply Theorem 4.7 to write *OBFair* as the conjunction of weak fairness of the actions described by each process's statements other than its *ncs* statement. That A3 is satisfied for the four actions A_{LM} in Figure 5.3 follows from conditions R3a, R3c, R6, and R7 of Section 5.4.2.

This leaves condition A1 for the actions. A1.1 is proved by using the type correctness invariant implied by Inv_{LM} to show that $\mathbb{E}\langle A_{LM}\rangle_{v_{LM}}$ equals $\mathbb{E}(A_{LM})$, and then substituting $pcBar_{OB}$ for pc_{LM} and $semBar_{OB}$ for sem_{LM} in $\mathbb{E}(A_{LM})$. For our example, this actually shows that $\overline{Inv_{LM}}$ implies $\overline{\mathbb{E}\langle A_{LM}\rangle_{v_{LM}}} \equiv Q_{OB}$ for all the actions A_{LM}. A1.2 is trivially satisfied for $CS_{LM}(p)$ and $Exit_{LM}(p)$, since Q_{OB} and P_{OB} are equal. The interesting conditions are A1.2 for $Wait_{LM}(0)$ and $Wait_{LM}(1)$. They are the kind of leads-to property we saw how to prove in Section 4.2.5. In fact, we now obtain a proof of A1.2 for $Wait_{LM}(0)$ by a simple modification of the proof in Section 4.2.5.3 that *OB* implies:

$$(5.18)\quad (pc(0) \in \{wait, w2\}) \leadsto (pc(0) = cs)$$

Let's drop the subscript *OB*, so the variables in any formula whose name has no subscript are the variables of *OB*. The proof of (5.18) is described by the proof lattice of Figures 4.4 and 4.5. A $\Box$ formula in a label on a box in a proof lattice means that the formula is conjoined to each formula inside the box. Since $F \leadsto G$ implies $(\Box H \land F) \leadsto (\Box H \land G)$ for any F, G, and H, we obtain a valid proof lattice (one whose leads-to assertions are all true) by conjoining $\Box\overline{Inv_{LM}} \land OBFair \land \Box Q$ to the labels of the outer boxes in the lattices of Figures 4.4 and 4.5. This makes those labels equal to $OBB \land \Box Q$. Since Q implies $pc(0) \in \{wait, w2\}$, we obtain a valid proof lattice by replacing the source node of the lattice in Figure 4.4 by $\Box Q$. Moreover, since the new label's conjunct $\Box Q$ implies $\Box(pc(0) \neq cs)$, so it's impossible for $pc(0)$ ever to equal cs, we can remove the sink node $pc(0) = cs$ and the edges to and from it from the lattice of Figure 4.5.[3] Since the label on the inner box containing $\Box\neg x(1)$, which is the new sink node, implies $\Box(pc(0) = w2)$, we now have a valid proof lattice that shows:

$$\models OBB \;\Rightarrow\; (\Box Q \;\leadsto\; \Box((pc(0) = w2) \land \neg x(1)))$$

This proves A1.2 for the action $Wait_{LM}(0)$.

To prove A1.2 for action $Wait_{LM}(1)$, it suffices to assume *OBB* and prove $\Box Q \leadsto \Box P$ for the formulas P and Q given in Figure 5.3 for the action. Here

[3]Equivalently, we can remove edge 8 and add an edge from $pc(0) = cs$ to FALSE and an edge from FALSE to $\Box\neg x(1)$, since FALSE implies anything.

is the proof sketch, which uses without mention some simple temporal logic, including transitivity of $\leadsto$.

1. $\Box Q \;\Rightarrow\; \Box\neg x(0)$
   1.1. $\Box Q \;\Rightarrow\; \Box(pc(0) \notin \{wait, w2\})$
   Proof: We proved in Section 4.2.5.3 that $pc(0) \in \{wait, w2\}$ leads to $pc(0) = cs$, and $\Box Q$ implies $\Box(pc(0) \neq cs)$.
   1.2. $\Box Q \wedge \Box(pc(0) \notin \{wait, w2\}) \;\Rightarrow\; \Box(pc(0) = ncs)$
   Proof: Q implies $pc(0) \notin \{cs, exit\}$, which by Inv and $pc(0) \notin \{wait, w2\}$ implies $pc(0) = ncs$.
   1.3. Q.E.D.
   Proof: By steps 1.1 and 1.2, since $Inv \wedge Q$ imply $pc(0) = ncs$, and Inv and $pc(0) = ncs$ imply $\neg x(0)$.
2. $\Box Q \wedge \Box\neg x(0) \;\leadsto\; \Box P$
   2.1. $\Box Q \wedge \Box\neg x(0) \;\leadsto\; (pc(1) = w2)$
   Proof: Q implies $pc(1) \in \{wait, w2, w3, w4\}$, and a straightforward proof using fairness of $PNext(1)$ and $\Box\neg x(0)$ shows
   $$(pc(1) \in \{wait, w2, w3, w4\}) \;\leadsto\; (pc(1) = w2)$$
   2.2. $\Box Q \wedge \Box\neg x(0) \wedge (pc(1) = w2) \;\Rightarrow\; \Box(pc(1) = w2)$
   Proof: $\Box Q$ implies $\Box(pc(1) \neq cs)$, and $(pc(1) = w2) \wedge \Box[Next]_v \wedge \Box(pc(1) \neq cs)$ implies $\Box(pc(1) = w2)$.
   2.3. Q.E.D.
   Proof: Steps 2.1 and 2.2 imply $\Box Q \wedge \Box\neg x(0) \leadsto \Box(pc(1) = w2)$, and $\Box P$ equals $\Box(pc(1) = w2) \wedge \Box\neg x(0)$.
3. Q.E.D.
   Proof: By steps 1 and 2,

5.4.4 A Closer Look at $\mathbb{E}$

5.4.4.1 A Syntactic View

Section 4.2.1 explained $\mathbb{E}$ semantically, defining $\mathbb{E}(A)$ to be true of a state s iff there exists a state t such that action A is true of the step $s \rightarrow t$. We now translate this semantic definition into a syntactic definition of $\mathbb{E}(A)$. A state is an assignment of values to variables, so we can restate that definition as:

E1. $\mathbb{E}(A)$ is true for an assignment of values to the unprimed variables iff there exists an assignment of values to the primed variables that makes A true.

A state predicate is true of a state iff it is true when its variables have the values assigned to them by the state. We can restate E1 as:

E2. $\mathbb{E}(A)$ is true (of a state) iff there exist values of the primed variables for which A is true.

We will now translate E2 into a precise syntactic definition of $\mathbb{E}$.

To do this, for any variable x, we now regard x and x' as two unrelated symbols. For an expression *exp*, we take *exp*$'$ to be the expression obtained by priming all the variables in *exp*. If *exp* contains a defined symbol whose definition contains variables, then all the variables in that definition are primed in *exp*$'$.

We now define AWITH to be substitution like WITH, except regarding x' as being a different variable from x. For example, if x, y, and z are variables, then:

$$\begin{array}{lll} (x' = x + 1) \text{ WITH } x \leftarrow y - z & \text{equals} & (y - z)' = (y - z) + 1 \\ (x' = x + 1) \text{ AWITH } x \leftarrow y - z & \text{equals} & x' = (y - z) + 1 \\ (x' = x + 1) \text{ AWITH } x' \leftarrow y - z & \text{equals} & (y - z) = x + 1 \end{array}$$

If *sym* is a defined symbol, then

$$(x' = x + sym) \text{ AWITH } x' \leftarrow y - z$$

equals

$$(y - z) = x + (sym \text{ AWITH } x' \leftarrow y - z)$$

If $sym \triangleq \sqrt{2 * x'}$, then this equals

$$(y - z) = x + \sqrt{2 * (y - z)}$$

Now let A be an action and let $x_1, \ldots, x_n$ be all the variables that appear in A. We can then write E2 as:

(5.19) $\mathbb{E}(A) \triangleq \exists\, c_1, \ldots, c_n : (A \text{ AWITH } x'_1 \leftarrow c_1, \ldots, x'_n \leftarrow c_n)$

Thus, we obtain $\mathbb{E}(A)$ from A by replacing the primed variables by bound constants that are existentially quantified. We informally describe this definition by saying that $\mathbb{E}(A)$ is obtained from A by existentially quantifying its primed variables.

5.4.4.2 Computing $\mathbb{E}$

The syntactic definition (5.19) of $\mathbb{E}$ immediately provides rules for writing $\mathbb{E}(A)$ in terms of formulas $\mathbb{E}(B_i)$, for B_i subactions of A. From the rule

$$\models (\exists\, c \,:\, A \vee B) \;\equiv\; (\exists\, c \,:\, A) \vee (\exists\, c \,:\, B)$$

we have

$\mathbb{E}$1. $\models \mathbb{E}(A \vee B) \;\equiv\; \mathbb{E}(A) \vee \mathbb{E}(B)$

For example, in program LM defined in Section 4.2.6.1, the next-state action $PNext(p)$ is the disjunction of actions $Ncs(p)$, $Wait(p)$, $Cs(p)$, and $Exit(p)$. Therefore, rule $\mathbb{E}$1 implies

$$\mathbb{E}(PNext(p)) \;\equiv\; \mathbb{E}(Ncs(p)) \vee \mathbb{E}(Wait(p)) \vee \mathbb{E}(Cs(p)) \vee \mathbb{E}(Exit(p))$$

The generalization of $\mathbb{E}$1 is:

$\mathbb{E}$2. $\models \mathbb{E}(\exists\, i \in S : A_i) \;\equiv\; \exists\, i \in S : \mathbb{E}(A_i)$

where S is a constant or state expression.

Another rule of existential quantification is that if the constant c does not occur in A, then $\exists\, c : (A \wedge B)$ is equivalent to $A \wedge (\exists\, c : B)$. From this we deduce:

$\mathbb{E}$3. If no variable appears primed in both A and B, then $\models \mathbb{E}(A \wedge B) \;\equiv\; \mathbb{E}(A) \wedge \mathbb{E}(B)$.

For example, in program LM we have:

$$\begin{aligned} Wait(p) \;\triangleq\; & \wedge\, (sem = 1) \wedge (pc(p) = wait) \\ & \wedge\, sem' = 0 \\ & \wedge\, pc' = (pc \text{ EXCEPT } p \mapsto cs) \end{aligned}$$

Therefore, rule $\mathbb{E}$3 implies

$$\begin{aligned} \mathbb{E}(Wait(p)) \;\equiv\; & \wedge\, \mathbb{E}((sem = 1) \wedge (pc(p) = wait)) \\ & \wedge\, \mathbb{E}(sem' = 0) \\ & \wedge\, \mathbb{E}(pc' = (pc \text{ EXCEPT } p \mapsto cs)) \end{aligned} \tag{5.20}$$

The following two rules also follow easily from (5.19) and properties of existential quantification:

$\mathbb{E}$4. If P is a state predicate, then $\models \mathbb{E}(P) \equiv P$.

$\mathbb{E}5$. If x is a variable and *exp* is a state expression, then $\models \mathbb{E}(x' = exp) \equiv$ TRUE.

From (5.20), $\mathbb{E}4$, and $\mathbb{E}5$, we deduce that $\mathbb{E}(Wait(p))$ equals $(sem = 1) \land (pc(p) = wait)$. Here is another obvious rule, which can be considered a generalization of $\mathbb{E}5$, since $c = exp$ is equivalent to $c \in \{exp\}$:

$\mathbb{E}6$ If x is a variable and *exp* is a state expression, then $\models \mathbb{E}(x' \in exp) \equiv (exp \neq \{\})$.

Rules $\mathbb{E}1$–$\mathbb{E}6$ are sufficient for computing $\mathbb{E}(A)$ for almost all subactions A that, like $PNext(p)$, appear in the definition of a program's next-state action. However, the definition of fairness does not contain such formulas $\mathbb{E}(A)$. Instead, it contains formulas of the form $\mathbb{E}\langle A\rangle_v$, which equals $\mathbb{E}(A \land (v' \neq v))$. None of those rules apply to such a formula. In particular, $\mathbb{E}3$ does not apply because v is the tuple of all the program's variables.

Most of the time, a subaction A in the definition of a program's next-state action does not allow stuttering steps. Therefore, $\langle A\rangle_v$ equals A, so $\mathbb{E}\langle A\rangle_v$ equals $\mathbb{E}(A)$ and we can apply the rules. For example, $\mathbb{E}\langle PNext(p)\rangle_v$ equals $\mathbb{E}(PNext(p))$ because a $PNext(p)$ step changes the value of $pc(p)$, so it can't be a stuttering step. We are using the substitutivity rule (3.34) of ordinary math to deduce

$$\models (A \equiv B) \;\Rightarrow\; (\mathbb{E}(A) \equiv \mathbb{E}(B))$$

(Even though substitutivity is not valid for TLA or the Logic of Actions, we can apply it to the syntactic definition (5.19) of $\mathbb{E}$, which treats x and x' as two different variables of ordinary math—that is, two different constants.)

However, we can't deduce $PNext(p) \equiv \langle PNext(p)\rangle_v$ from the definition of $PNext(p)$. For example, if $pc(p) = cs$, then the definition of $PNext(p)$ asserts

$$pc' = (pc \text{ EXCEPT } p \mapsto exit)$$

If p is not in the domain of pc, then $pc'(p) = pc(p)$. If pc is not a function, then we have no idea what $pc'(p)$ equals, so it could equal $pc(p)$. Fortunately, we care what $\mathbb{E}\langle PNext(p)\rangle_v$ equals only in reachable states of LM. So, we just have to prove that Inv implies $\mathbb{E}\langle PNext(p)\rangle_v \equiv \mathbb{E}(PNext(p))$ for an invariant Inv of LM that asserts type correctness. To do this, we observe that for any action A and state predicate P, rules $\mathbb{E}3$ and $\mathbb{E}4$ imply $P \land \mathbb{E}(A) \equiv \mathbb{E}(P \land A)$. So, to prove that Inv implies $\mathbb{E}\langle PNext(p)\rangle_v \equiv \mathbb{E}(PNext(p))$, it suffices to prove

$$\models Inv \;\Rightarrow\; (\langle PNext(p)\rangle_v \;\equiv\; PNext(p))$$

which is straightforward. In general, we reason about liveness under the assumption that the program's safety property is satisfied, so we can assume $\Box \mathit{Inv}$ is true for an invariant Inv of the program.

Since the formula $\Box[\mathit{Next}]_v$ always allows stuttering steps, there is no need for a next-state action Next to allow them. Usually, it doesn't. However, there is no reason for Next not to allow stuttering steps, and sometimes it's more convenient to write a subaction A that allows them. In that case, we have to use the definition (5.19) to compute $\mathbb{E}\langle A\rangle_v$. However, we apply the definition to $\mathbb{E}(\mathit{Inv} \land \langle A\rangle_v)$, which equals $\mathbb{E}\langle \mathit{Inv} \land A\rangle_v$, for a program invariant Inv.

5.4.4.3 The Trouble With $\mathbb{E}$

Refinement is based on substitution. Program OB refines LM means:

$$(5.21)\quad \models OB \;\Rightarrow\; (LM \text{ WITH } pc \leftarrow \mathit{pcBar},\ \mathit{sem} \leftarrow \mathit{semBar})$$

We no longer need the subscripts that were added to help us understand which program an identifier refers to. We continue using the abbreviation that, for any formula F:

$$\overline{F} \;\triangleq\; (F \text{ WITH } pc \leftarrow \mathit{pcBar},\ \mathit{sem} \leftarrow \mathit{semBar})$$

Almost without thinking, we replaced $\overline{\mathit{Init} \land \Box[\mathit{Next}]_v}$ with the equivalent property $\overline{\mathit{Init}} \land \Box[\overline{\mathit{Next}}]_{\overline{v}}$. We were actually using these three rules:

- $\overline{F \land G} \equiv \overline{F} \land \overline{G}$ for any formulas F and G.
- $\overline{\Box F} \equiv \Box\overline{F}$ for any formula F.
- $\overline{[A]_v} \equiv [\overline{A}]_{\overline{v}}$ for any action A and state expression v.

The first asserts that substitution *distributes over* $\lor$; the second asserts that substitution distributes over $\Box$; and the third asserts that substitution distributes over the construct $[\ldots]_{\ldots}$.

We expect substitution to distribute in this way over all mathematical operators, so we would expect $\overline{\mathbb{E}(A)}$ and $\mathbb{E}(\overline{A})$ to be equal for any action A. In fact, they are equal for most actions encountered in practice. But here's an action A for which they aren't for the refinement mapping of (5.21):

$$\begin{aligned} A \;\triangleq\; & \land\ pc' = (p \in \{0,1\} \mapsto \mathit{wait}) \\ & \land\ \mathit{sem}' = 0 \end{aligned}$$

Rules $\mathbb{E}3$ and $\mathbb{E}5$ imply that $\mathbb{E}(A)$ equals TRUE, so $\overline{\mathbb{E}(A)}$ equals TRUE. By definition of the refinement mapping:

$$\overline{A} \triangleq \begin{array}{l} \land\ pcBar' = (p \in \{0,1\} \mapsto wait) \\ \land\ semBar' = 0 \end{array}$$

$\overline{A}$ implies $pcBar'(p) = wait$ for $p \in \{0,1\}$. By definition of $pcBar$, this implies:

(1) $pc'(p) \in \{w2, w3, w4, wait\}$ for p equal to 0 or 1.

But $\overline{A}$ also implies $semBar' = 0$, which by the definition of $semBar$ implies:

(2) $pc'(p) \in \{cs, exit\}$ for p equal to 0 or 1.

Both (1) and (2) can't be true, so $\overline{A}$ must equal FALSE and thus $\mathbb{E}(\overline{A})$ equals FALSE. Therefore, $\overline{\mathbb{E}(A)}$ does not equal $\mathbb{E}(\overline{A})$, so substitution does not always distribute over $\mathbb{E}$.

The reason substitution doesn't distribute over $\mathbb{E}$ is that $\mathbb{E}(\overline{A})$ performs the substitutions $pc \leftarrow pcBar$ and $sem \leftarrow semBar$ for the primed variables pc' and sem'. However, as we see from (5.19), those primed variables do not occur in $\mathbb{E}(A)$; they are replaced by bound constants. The substitutions should be performed only on the unprimed variables. Therefore:

$$\mathbb{E}(A) \text{ WITH } pc \leftarrow \ldots, sem \leftarrow \ldots$$

does not equal

$$\mathbb{E}(A \text{ WITH } pc \leftarrow \ldots, sem \leftarrow \ldots)$$

Instead, it equals

$$\mathbb{E}(A \text{ AWITH } pc \leftarrow \ldots, sem \leftarrow \ldots)$$

which substitutes only for unprimed variables.

Since WF and SF are defined in terms of $\mathbb{E}$, substitution does not distribute over them either. We proved that OB refines $\overline{LM}$ by proving that OB implies $\overline{\text{WF}_v(A)}$ for six actions A. To evaluate $\overline{\text{WF}_v(A)}$, we expanded the definition of WF. Since substitution distributes over all the operators other than $\mathbb{E}$ in the definition of $\text{WF}_v(PN)$, including in the definition $PN(p)$, we could perform the substitutions everywhere in the resulting formula except in $\overline{\mathbb{E}\langle A\rangle_v}$. We could then have used (5.19) to expand the definition of $\mathbb{E}$ and perform the substitution on the resulting formula, which contains no primed variables. (This is equivalent to performing the substitution in $\mathbb{E}\langle A\rangle_v$, except using AWITH instead of WITH.)

While expanding the definition of $\mathbb{E}$ in this way would have allowed $\overline{\mathbb{E}\langle A\rangle_v}$ to be evaluated, it would have required applying $\mathbb{E}$ to an action that was more complicated than $\langle A\rangle_v$. That's not what we did in the proof sketch in Section 5.4.3. Instead we showed that $\langle A\rangle_v$ equals A and performed the substitution on $\mathbb{E}(A)$. Showing $\langle A\rangle_v \equiv A$ required an invariant *Inv* of *LM*, but because *OB* refines $\overline{LMSafe}$, the formula $\overline{Inv}$ is an invariant of *OB*, allowing us to deduce that $\overline{\mathbb{E}\langle A\rangle_v}$ equals $\overline{A}$.

Substitution not distributing over $\mathbb{E}$ makes $\mathbb{E}$ mathematically weird. You should be suspicious of such weird things. The operators $\Box$ and $'$ (prime) that TLA adds to ordinary math are weird because they are not substitutive. But substitution does distribute over them. Moreover, temporal logic is a well-studied field of math. I find $\mathbb{E}$ weirder than the temporal logic operators.

However, fairness is an important concept in concurrent programs. The WF and SF operators are the mathematical expressions of what fairness has meant since Dijkstra introduced the assumption of weak fairness in 1965 [9]. There seems to be no good way to express it mathematically without the operator $\mathbb{E}$.

A similarly weird operator has been at the heart of traditional programs since the earliest coding languages—namely, the action composition operator "$\cdot$" introduced in Section 3.4.1.4. If $x_1, \ldots, x_n$ are all the variables that appear in actions A and B, then $A \cdot B$ can be defined syntactically by:

$$\begin{aligned} A \cdot B \;\triangleq\; \exists\, c_1, \ldots, c_n : \;& \land\, (A \text{ AWITH } x'_1 \leftarrow c_1, \ldots, x'_n \leftarrow c_n) \\ & \land\, (B \text{ AWITH } x_1 \leftarrow c_1, \ldots, x_n \leftarrow c_n) \end{aligned}$$

The primed variables of A and the unprimed variables of B are replaced by bound constants, and substitution does not distribute over "$\cdot$" for the same reason it doesn't distribute over $\mathbb{E}$.

The common methods for reasoning about traditional programs written in an imperative language can be viewed as a form of Hoare logic. As explained in Appendix Section A.4, such a logic can be viewed mathematically as defining the meaning of a statement S to be an action A_S. If the meanings of statements S and T are the actions A_S and A_T, then the meaning of $S; T$ is the action $A_S \cdot A_T$.

With this way of reasoning, the semicolon of imperative coding languages therefore has the same weirdness as the $\mathbb{E}$ operator. I suspect this was never discovered because people thought of programs in terms of conventional code, and it makes no sense to implement a variable x by an expression when x can appear in an assignment statement $x := \ldots$.

5.5 A Warning

We have defined correctness of a program S to mean $\models S \Rightarrow P$ for some property P. We have to be careful to make sure that we have chosen P so that this implies what we really want correctness of the program to mean. As discussed in Section 4.3, we have to be concerned with the accuracy of P.

When correctness asserts that S refines a program T, the property P is (T WITH ...) for a refinement mapping "...". That refinement mapping is as important a part of the property as the program T, and it must be examined just as carefully to be sure that proving refinement means what you want it to. As an extreme example, *OB* also implements *LM* under this refinement mapping:

$$pc_{LM} \leftarrow (p \in \{0,1\} \mapsto ncs),\ \ sem_{LM} \leftarrow 1$$

Implementation under this refinement mapping tells us nothing about *OB*, because under it, every behavior of *OB* implements a behavior in which all processes remain forever in their noncritical sections. The program obtained by replacing the next-state action of *OB* by FALSE also implements *LM* under this refinement mapping.

Such an egregiously useless refinement mapping can often be detected because, under a refinement mapping that implements behaviors of a program T by behaviors of program S that do nothing, S won't implement the fairness properties of T. However, programs often don't require that actions representing the initiation of an operation by the environment ever occur. In such a case, it's a good idea to make sure that S refines T when fairness requirements are added to those actions in both programs. This is an application of the general idea of adding fairness to verify possibility that was introduced in Section 4.3.2.

Chapter 6

Auxiliary Variables

An auxiliary variable is a variable that is added to an abstract program without altering the values assumed by the program's regular variables. It's sometimes necessary to add auxiliary variables to a program in order to prove that it refines another program. Sections 6.2, 6.3, and 6.4 define the three kinds of auxiliary variables that may be needed, illustrating them with silly little examples. Section 6.5 describes a realistic example that makes use of all three kinds of auxiliary variables. We begin with a section that explains variable hiding, which is the basis for auxiliary variables and is also used in Chapter 7.

6.1 Variable Hiding

6.1.1 Introduction

Recall the behavior predicate F_{12}, discussed in Section 4.1.2, that is true of a behavior iff the value of x can equal 2 in a state only if x equaled 1 in a previous state. We gave a semantic definition of F_{12}; it can't be expressed in RTLA or TLA as those languages have been defined so far. We observed that F_{12} can be expressed as the abstract program S_{12}, defined in (4.2), by introducing an additional variable y.

The variable x that we're interested in is called an *interface* variable. The variable y that's used only to describe how the values of x can change is called an *internal* variable. There's a problem with using the internal variable y to describe F_{12}. Consider the abstract program S_x that starts with $x = 0$ and can keep incrementing x by one:

$$S_x \triangleq (x = 0) \wedge \Box[x' = x + 1]_x$$

Since S_x allows x to equal 2 only after it has equaled 1, it satisfies property F_{12}. However, S_x doesn't imply S_{12} because S_{12} describes how the values of x and y change, while S_x allows behaviors in which y can have any values.

We want to express F_{12} by a formula that asserts of a behavior σ that there is some way to assign values to y that makes S_{12} true, but says nothing about the actual values of y. As mentioned in Section 4.1.2, that formula is written $\boldsymbol{\exists}\, y : S_{12}$. The operator $\boldsymbol{\exists}$ is explained here.

In ordinary math, the formula $\exists\, y : x * y^2 = 36$ asserts that there is some value y for which $x * y^2$ equals 36, but says nothing about the actual value of y. The variables of ordinary math correspond to the constants of temporal logic. The y in $\exists\, y : S_{12}$ is a constant, so that formula asserts that there is a constant value y that satisfies S_{12}; and that value equals TRUE if the initial value of x is 1, otherwise it equals FALSE. Formula $\exists\, y : S_{12}$ asserts that x can never equal 2 unless the initial value of x is 1, which is not what F_{12} asserts.

The formula $\boldsymbol{\exists}\, y : S_{12}$ is true of a behavior iff the values of x in that behavior are the same as its values in a behavior satisfying S_{12}, where y is a variable rather than a constant; but it says nothing about the actual values assumed by y. Thus, y is not a (free) variable of $\boldsymbol{\exists}\, y : S_{12}$. The precise definition of $\boldsymbol{\exists}$ is subtle and is given below. For now, we just need to know that $[\![\boldsymbol{\exists}\, y : S_{12}]\!]$ equals F_{12}. I like to say that $\boldsymbol{\exists}\, y : S_{12}$ is formula S_{12} with y hidden, because $\boldsymbol{\exists}$ corresponds to what hiding seems to mean in coding languages.

We can now generalize abstract programs to allow quantification over variables. As with the operator $\exists$, we let $\boldsymbol{\exists}\, y_1, \ldots, y_n : F$ be an abbreviation for $\boldsymbol{\exists}\, y_1 : \ldots \boldsymbol{\exists}\, y_n : F$. The general form of an abstract program with hidden variables is:

(6.1) $\boldsymbol{\exists}\, y_1, \ldots, y_k \; : \; \mathit{Init} \wedge \Box[\mathit{Next}]_v \wedge L$

with internal (bound) variables $y_1, \ldots, y_k$. (The interface variables are the free variables of the formula.) Theorem 4.8 shows that any property that can be described mathematically can be written in this form, with a single bound variable. However, $\boldsymbol{\exists}$ is of little use in practice. The only role it plays is telling us that, when implementing the program, it doesn't matter how the internal variables are refined. That can be stated just as well in a comment; we don't need to introduce a new operator just for that. In fact, although $\boldsymbol{\exists}$ is an operator of TLA$^+$ and is recognized by the parser, none of the current tools handle it. Model checking formulas containing $\boldsymbol{\exists}$ seems to be computationally infeasible. I don't know of any engineer wanting to use it.

The reason to understand the temporal existential quantification operator $\boldsymbol{\exists}$ is that it is the logical underpinning of important concepts such as the auxiliary variables discussed in this chapter.

6.1.2 Reasoning About $\boldsymbol{\exists}$

Allowing an abstract program to be described with a formula of the form (6.1) raises the question of how to reason about such formulas. The answer is that the operator $\boldsymbol{\exists}$ obeys the rules for the quantifier $\exists$ of predicate logic given in Section 2.1.9.3, except with program variables (now called variables) replacing the mathematical variables (now called constants). There are two ways we want to reason about the formula (6.1): (1) show that it satisfies a property G, and (2) show that it is refined by a program T.

For (1), we use the $\boldsymbol{\exists}$ elimination rule (the analog of the $\exists$ elimination rule of predicate logic) when the bound variable does not occur in the formula being proved. If v does not occur (free) in G, then that rule asserts

$$\models F \Rightarrow G \quad \text{implies} \quad \models (\boldsymbol{\exists}\, v : F) \Rightarrow G$$

Therefore, if none of the variables y_i occur in G, by applying the rule multiple times we prove

$$\models (\boldsymbol{\exists}\, y_1, \ldots, y_k : S) \Rightarrow G$$

by proving $\models S \Rightarrow G$, treating the internal variables y_i and the interface variables the same. If y_i occurs free in G, then we can replace it in the quantified formula by any variable that does not occur in G or S.

For (2), applying the $\boldsymbol{\exists}$ Introduction rule multiple times, we prove

(6.2) $\models\ T \ \Rightarrow\ \boldsymbol{\exists}\, y_1, \ldots, y_k : S$

for a program T by proving:

(6.3) $\models\ T \ \Rightarrow\ (S \text{ WITH } y_1 \leftarrow \mathit{exp}_1, \ldots, y_k \leftarrow \mathit{exp}_k)$

for expressions exp_i. For every interface variable x of S, which in practice must also occur in T, the WITH clause includes an implicit substitution $x \leftarrow x$ that substitutes the variable x of T for the variable x of S. Thus, the WITH clause describes a refinement mapping under which T refines S.

This raises a question: If (6.2) is true, do there always exist expressions exp_i for which (6.3) is true? The answer is no, if we can use only the variables that appear in T to define the refinement mapping. If S has the form $\mathit{Init} \land \Box[\mathit{Next}]_v \land L$, then the answer is yes if we're allowed to add auxiliary variables to T. Adding an auxiliary variable a (which does not occur in T) to T means writing a formula T^a such that $\exists\, a : T^a$ is equivalent to T. By this equivalence, we can verify $\models T \Rightarrow S$ by verifying $\models (\exists\, a : T^a) \Rightarrow S$. By the $\exists$ Elimination rule, we do this by verifying $\models T^a \Rightarrow S$. And to verify this, we can use a as well as the variables of T to define the refinement mapping. Auxiliary variables are the main topic of this chapter and are discussed below.

6.1.3 The Definition

The standard way temporal existential quantification is defined in most temporal logics is not suitable for TLA because it does not preserve stuttering insensitivity (SI), defined in Section 3.5.3. It's the natural way to define it for RTLA, so we will call the operator defined in that way $\exists_{\text{RTLA}}$.

To define $\exists_{\text{RTLA}}$, we first define $s =_y t$ to be true for states s and t iff the values of all variables except y are the same in states s and t. Remembering that $\sigma(i)$ is state number i of a behavior σ, we define the relation $\simeq_y$ on behaviors by:

$$\sigma \simeq_y \tau \;\triangleq\; \forall\, i \in \mathbb{N} : \sigma(i) =_y \tau(i)$$

Therefore, $\sigma \simeq_y \tau$ asserts that behaviors σ and τ are the same except for the values assigned to y by their states. We then define $\exists_{\text{RTLA}}\, y : F$ to be satisfied by a behavior σ iff it is satisfied by some behavior τ with $\sigma \simeq_y \tau$.

The operator $\exists_{\text{RTLA}}$ is not a suitable hiding operator for properties, and hence not suitable for TLA, because the formula $\exists_{\text{RTLA}}\, y : F$ need not be a property even if F is. For example, let F be the following formula, where $\lfloor r \rfloor$ is the largest integer less than or equal to r:

(6.4) $(x = y = 0) \wedge \Box[(y' = y + 1) \wedge (x' = \lfloor y'/2 \rfloor)]_{\langle x, y \rangle}$

Ignoring the values of other variables, the property F is satisfied by this non-halting behavior with no stuttering steps:

$$\begin{bmatrix} x :: 0 \\ y :: 0 \end{bmatrix}_0 \rightarrow \begin{bmatrix} x :: 0 \\ y :: 1 \end{bmatrix}_1 \rightarrow \begin{bmatrix} x :: 1 \\ y :: 2 \end{bmatrix}_2 \rightarrow \begin{bmatrix} x :: 1 \\ y :: 3 \end{bmatrix}_3 \rightarrow \begin{bmatrix} x :: 2 \\ y :: 4 \end{bmatrix}_4 \rightarrow \cdots$$

The non-halting behaviors of $\exists_{\text{RTLA}}\, y : F$ consist of this behavior:

(6.5) $\begin{bmatrix} x :: 0 \end{bmatrix}_0 \rightarrow \begin{bmatrix} x :: 0 \end{bmatrix}_1 \rightarrow \begin{bmatrix} x :: 1 \end{bmatrix}_2 \rightarrow \begin{bmatrix} x :: 1 \end{bmatrix}_3 \rightarrow \begin{bmatrix} x :: 2 \end{bmatrix}_4 \rightarrow \cdots$

and behaviors obtained from it by adding stuttering steps. An SI formula containing the one free variable x that allows behavior (6.5) should also allow this behavior:

$$(6.6)\quad \Big[x :: 0\Big]_0 \to \Big[x :: 1\Big]_1 \to \Big[x :: 2\Big]_2 \to \Big[x :: 3\Big]_3 \to \Big[x :: 4\Big]_4 \to \cdots$$

which $\boldsymbol{\exists}_{\text{RTLA}}\, y : F$ does not allow, so it is not a property.

To obtain the proper quantifier $\boldsymbol{\exists}$ for TLA, we modify the definition of $\boldsymbol{\exists}_{\text{RTLA}}$ so $\boldsymbol{\exists}\, y : F$ is satisfied by (6.6). The definition of $\boldsymbol{\exists}$ is the same as that of $\boldsymbol{\exists}_{\text{RTLA}}$ except with the relation $\simeq_y$ on behaviors replaced by a relation $\sim_y$. This relation is defined so $\sigma \sim_y \tau$ means approximately that σ can be obtained from τ by adding and removing stuttering steps and then changing the values of y. The precise definition of $\sim_y$ is a bit subtle. (In fact, its definition in [34] is wrong.) To define $\sim_y$, we first define the operator $\natural_y$ on behaviors. This operator is the same as the operator $\natural$ defined in (3.37) except with "$=$" replaced by "$=_y$". Thus, $\natural_y$ removes "almost stuttering" steps instead of just stuttering steps, where a step $s \to t$ is almost stuttering if $s =_y t$. Here's the precise definition:

```
$\natural_y(\sigma) \;\triangleq\;$ IF $\forall\, i \in \mathbb{N} : \sigma(i) =_y \sigma(0)$
                 THEN $\sigma$
                 ELSE IF $\sigma(0) =_y \sigma(1)$ THEN $\natural_y(\mathit{Tail}(\sigma))$
                                          ELSE $\langle \sigma(0) \rangle \circ \natural_y(\mathit{Tail}(\sigma))$
```

We now define $\sigma \sim_y \tau$ to equal $\natural_y(\sigma) \simeq_y \natural_y(\tau)$ and define $\boldsymbol{\exists}\, y : F$ to be satisfied by a behavior F iff there is a behavior τ satisfying F such that $\sigma \sim_y \tau$. Observe that $\sigma \simeq_y \tau$ implies $\sigma \sim_y \tau$. Therefore, $\boldsymbol{\exists}_{\text{RTLA}}\, y : F$ implies $\boldsymbol{\exists}\, y : F$ for any behavior predicate F.

One reason not to use $\boldsymbol{\exists}$ is that if S is a safety property, then $\boldsymbol{\exists}\, y : S$ need not be a safety property. Temporal quantification destroys the nice separation of safety and liveness provided by our way of describing abstract programs. For example, let F be this safety property for an abstract program:

$$\begin{aligned}(6.7)\quad &\wedge\ (x = 0) \wedge (y \in \mathbb{N}) \\ &\wedge\ \Box[(y > 0) \wedge (x' = x + 1) \wedge (y' = y - 1)]_{\langle x, y \rangle}\end{aligned}$$

In a behavior satisfying this formula, x cannot be incremented forever because eventually y would equal 0, making any further non-stuttering steps impossible. Therefore, formula $\boldsymbol{\exists}\, y : F$ is equivalent to

$$(6.8)\quad (x = 0) \wedge \Box[x' = x + 1]_x \wedge \Diamond\Box[x' = x]_x$$

To see that this is not a safety property, remember that a behavior σ satisfies a safety property iff every finite prefix of σ satisfies that property. Consider a behavior σ in which x does keep being incremented forever. Every finite prefix of σ satisfies (6.8), since completing the prefix with stuttering steps makes the behavior satisfy the liveness property $\Diamond\Box[x' = x]_x$. However, σ doesn't satisfy (6.8) because it doesn't satisfy this liveness property. Therefore, even though formula F, defined to equal (6.7), is a safety property, formula $\exists\, y : F$, which is equivalent to (6.8), is not a safety property.

6.2 History Variables

The simplest kind of auxiliary variable is a history variable. As the name implies, a history variable is used to remember things that happened in the past and can't be deduced from the current state. We may need to add a history variable to T to prove $\models T \Rightarrow S$ when the internal state of S records information about past events that isn't needed to describe the behavior of its interface variables.

6.2.1 How to Add a History Variable

Except in one unusual case described in Section 6.3.5, we add an auxiliary variable to an abstract program by adding it to the safety part of the program. Thus, if T equals $\mathit{Init} \land \Box[\mathit{Next}]_v \land L$ for a liveness property L, then the formula T^h obtained by adding a history variable h will equal

$$\mathit{Init}^h \land \Box[\mathit{Next}^h]_{vh} \land L$$

where Init^h and Next^h are obtained by augmenting Init and Next to describe, respectively, the initial value of h and how h can change; and vh is the tuple $v \circ \langle h \rangle$ of the variables of v and the variable h. Since h does not appear in L, the formula $\exists\, h : T^h$ equals

$$(\exists\, h : \mathit{Init}^h \land \Box[\mathit{Next}^h]_{vh}) \;\land\; L$$

We can therefore ignore L for now, so we assume T equals $\mathit{Init} \land \Box[\mathit{Next}]_v$ and show how to define Init^h and Next^h.

We use a tiny example to illustrate history variables. There is an abstract program in which a user inputs a sequence of real numbers and the system displays the average of the numbers entered thus far. The interface variables are inp and avg. Initially, inp equals a value rdy that is not a number and $\mathit{avg} = 0$. The user's input action sets inp to a real number and leaves avg

$$
\begin{aligned}
&\mathit{InitS} \triangleq (\mathit{inp} = \mathit{rdy}) \land (\mathit{avg} = 0) \land \mathit{seq} = \langle\,\rangle \\
&\mathit{User} \triangleq \begin{array}{l} \land\ \mathit{inp} = \mathit{rdy} \\ \land\ \mathit{inp}' \in \mathbb{R} \\ \land\ (\mathit{avg}' = \mathit{avg}) \land (\mathit{seq}' = \mathit{seq}) \end{array} \\
&\mathit{Syst} \triangleq \begin{array}{l} \land\ \mathit{inp} \in \mathbb{R} \\ \land\ \mathit{seq}' = \mathit{Append}(\mathit{seq}, \mathit{inp}) \\ \land\ \mathit{avg}' = \mathit{SeqSum}(\mathit{seq}') \,/\, \mathit{Len}(\mathit{seq}') \\ \land\ \mathit{inp}' = \mathit{rdy} \end{array} \\
&\mathit{NextS} \triangleq \mathit{User} \lor \mathit{Syst} \\
&\mathit{IS} \triangleq \mathit{InitS} \land \Box[\mathit{NextS}]_{\langle \mathit{inp}, \mathit{avg}, \mathit{seq} \rangle} \\
&S \triangleq \boldsymbol{\exists}\, \mathit{seq} : \mathit{IS}
\end{aligned}
$$

Figure 6.1: The abstract averaging program S.

unchanged. The system's output action sets *avg* to the new average of the inputs and resets *inp* to *rdy*.

This abstract program is described by formula S of Figure 6.1. It uses an internal variable *seq* whose value is the ordinal sequence of numbers input so far. Recall that $\mathbb{R}$ is the set of real numbers, and Section 2.3.2 defines these operators on sequences *seq*: $\mathit{Append}(\mathit{seq}, \mathit{inp})$ is the sequence obtained by appending *inp* to the end of *seq*; $\mathit{Len}(\mathit{seq})$ is the length of *seq*; and $\mathit{Tail}(\mathit{seq})$ is the sequence obtained by removing the first element of *seq* if *seq* is nonempty. The operator *SeqSum* is defined as follows so that $\mathit{SeqSum}(\mathit{sq})$ is the sum of the elements of a finite sequence *sq* of numbers:

$$\mathit{SeqSum}(\mathit{sq}) \triangleq \textsc{if}\ \mathit{sq} = \langle\,\rangle\ \textsc{then}\ 0\ \textsc{else}\ \mathit{sq}(1) + \mathit{SeqSum}(\mathit{Tail}(\mathit{sq}))$$

Using the internal variable *seq* to write the behavior predicate S is arguably the clearest way to describe the values assumed by the interface variables *inp* and *avg*. It's a natural way to explain that the value of *avg* is the average of the values that have been input. However, it's not a good way to describe how to implement the system. There's no need for an implementation to remember the entire sequence of past inputs; it can just remember the number of inputs and their sum. In fact, it doesn't even need an internal variable to remember the sum. We can implement it with an abstract program T that implements S using only a single internal variable *num* whose value is the number of inputs that the user has entered.

```
variables $inp = rdy,\ avg = 0,\ num = 0$ ;
while TRUE do
  $usr$: $inp \ :\in \mathbb{R}$ ;
  $sys$: $avg \ := (avg * num + inp) \,/\, (num + 1)$ ;
         $num \ := num + 1$ ;
         $inp \ := rdy$
end while
```

Figure 6.2: Abstract program T in pseudocode.

We first describe T in pseudocode and construct T^h by adding a history variable h to the code. The TLA translations of the pseudocode show how to add a history variable to an abstract program described in TLA.

It's natural to think of the user and the system in this example as two separate processes. However, the abstract programs S and T are predicates on behaviors, which are mathematical objects. *Process* is not a mathematical concept; it's a way in which we interpret predicates on behaviors. For simplicity, we write T as a single-process program.

The pseudocode for program T is in Figure 6.2. It uses the operator $:\in$ introduced in Figure 4.8, so statement *usr* sets *inp* to an arbitrary element of $\mathbb{R}$. Since we're not concerned with implementing T, there's no reason to hide its internal variable *num*.

Because the sum of n numbers whose average is a is $n * a$, it should be clear that program T implements program S. But showing that T implements S requires defining a refinement mapping under which T implements *IS* (program S without variable *seq* hidden). And that requires adding an auxiliary variable that records the sequence of input values. Adding the required auxiliary variable h is simple and obvious. We just add the two pieces of code shown in black in Figure 6.3.

It is a straightforward exercise to prove

$$\models T^h \Rightarrow (IS \text{ WITH } inp \leftarrow inp,\ avg \leftarrow avg,\ seq \leftarrow h)$$

using that fact that

$$avg \ \equiv \ \text{IF } \ h = \langle\,\rangle \ \text{ THEN } \ 0 \ \text{ ELSE } \ SeqSum(h) \,/\, Len(h)$$

is an invariant of T^h. To show that this proves $\models T \Rightarrow S$, we have to show that T^h actually is obtained by adding the auxiliary variable h to T—that is, we have to show that T is equivalent to $\exists\, h : T^h$. This requires showing (i) $\models (\exists\, h : T^h) \Rightarrow T$ and (ii) $\models T \Rightarrow (\exists\, h : T^h)$.

```
**variables** $inp = rdy,\ avg = 0,\ num = 0,\ h = \langle\,\rangle$ ;
**while** TRUE **do**
  $usr$: $inp :\in \mathbb{R}$ ;
  $sys$: $avg := (avg * num + inp) / (num + 1)$ ;
         $num := num + 1$ ;
         $h := Append(h, inp)$ ;
         $inp := rdy$
**end while**
```

Figure 6.3: Abstract program T^h in pseudocode.

To show (i) we have to show $\models T^h \Rightarrow T$, which is obvious because it's easy to see that the initial predicate and next-state action of T^h imply the initial predicate and next-state action of T. To show (ii), we have to show that for any behavior σ satisfying T there is a behavior τ satisfying T^h with $\tau \sim_h \sigma$. From the code for T^h, it's easy to obtain a recursive definition of the value of h in each state $\tau(i)$ of τ. The declaration of h provides the value of h in state $\tau(0)$, and the rest of the code defines the value of h in state $\tau(i+1)$ as a function of its value and the value of pc in state $\tau(i)$.

It's pretty obvious how to generalize from this example to adding a history variable h to any abstract program T described by pseudocode. We let the initial value of h be any expression that can contain the variables of T. We modify the pseudocode by adding at most one statement assigning a value to h to any action of the code. The right-hand side of the assignment can contain h as well as the variables of T. Making this precise would require making pseudocode precise, which we don't want to do. When we want to be precise, we use math.

So, let's now see how we add a history variable when the abstract program T is written in TLA. The translation of the code in Figure 6.2 to TLA defines

$$T \ \triangleq\ Init \wedge \Box[Next]_{\langle inp, avg, num\rangle} \quad \textbf{where} \quad Next \ \triangleq\ Usr \vee Sys$$

Actions Usr and Sys are the actions executed from control points usr and sys, respectively. The TLA translation of the code in Figure 6.3 is

(6.9)
$$\begin{aligned} T^h \ &\triangleq\ Init^h \wedge \Box[Next^h]_{\langle inp, avg, num, h\rangle} \\ \textbf{where}\ \ Init^h &\triangleq Init \wedge (h = \langle\,\rangle) \\ Next^h &\triangleq Usr^h \vee Sys^h \\ Usr^h &\triangleq Usr \wedge (h' = h) \\ Sys^h &\triangleq Sys \wedge (h' = Append(h, inp)) \end{aligned}$$

Here is the general result that describes how to add a history variable to a program. Its proof is a simple generalization of the proof for our example.

Theorem 6.1 (History Variable) Let T equal $\mathit{Init} \land \Box[\mathit{Next}]_v$, where Next equals $\exists\, i \in I : A_i$ and v is the tuple of variables in T, and assume h is not one of those variables. If T^h equals $\mathit{Init}^h \land \Box[\mathit{Next}^h]_{vh}$, where

- $\mathit{Init}^h \;\triangleq\; \mathit{Init} \land (h = \mathit{exp})$
- $\mathit{Next}^h \;\triangleq\; \exists\, i \in I \,:\, A_i \land (h' = \mathit{exp}_i)$
- $\mathit{vh} \;\triangleq\; v \circ \langle h \rangle$
- exp is a state expression that does not contain the variable h, and the exp_i are step expressions that do not contain h',

then $\models T \;\equiv\; \boldsymbol{\exists}\, h : T^h$.

6.2.2 History Variables and Fairness

We add a history variable h to a safety property T of the form $\mathit{Init} \land \Box[\mathit{Next}]_v$ to obtain a formula T^h such that $\boldsymbol{\exists}\, h : T^h$ is equivalent to T. If a program also contains a liveness condition L, this gives us the program $T^h \land L$. Since the variable h does not occur in L, the formula $\boldsymbol{\exists}\, h : T^h \land L$ is equivalent to $(\boldsymbol{\exists}\, h : T^h) \land L$ which equals $T \land L$. Therefore the history variable h is an auxiliary variable for $T^h \land L$.

As explained in Section 4.2.7, the standard form for the liveness condition of a program is the conjunction of weak and/or strong fairness conditions of subactions of its next-state action. Even if $T \land L$ has this form, $T^h \land L$ will not because a subaction of Next will not be a subaction of Next^h. (An action that does not mention h cannot imply Next^h.) This means that we can't apply Theorem 4.6 to show that $\langle\, T^h, L \rangle$ is machine closed. However, we can show as follows that if $\langle\, T, L \rangle$ is machine closed, then $\langle\, T^h, L \rangle$ is also machine closed. By definition of machine closure, this means showing that any finite behavior ρ satisfying T^h can be extended to an infinite behavior satisfying $T^h \land L$. Since T^h implies T, machine closure of $\langle\, T, L \rangle$ implies ρ can be extended to a behavior $\rho \circ \sigma$ satisfying $T \land L$. By definition of T^h, we can modify the values of h in the states of σ to obtain a behavior such that $\rho \circ \tau$ satisfies T^h. Since the truth of L does not depend on the values of h, the behavior $\rho \circ \tau$ also satisfies L, as required.

When using TLA, the fact that L will contain fairness conditions on actions that are not subactions of Next^h makes no difference. However, not everyone uses TLA. In some approaches, abstract programs are described in something like a coding language, and they define fairness only in terms of weak and strong fairness of subactions of the next-state action. So, it is interesting to know if we can replace a fairness condition on a subaction B_i of T with the same fairness condition on a corresponding subaction B_i^h of T^h. We can, under the following condition, which is likely to be satisfied by programs written in those other languages: The next-state action of T must be the disjunction of actions A_i, and each B_i must be a subaction of A_i such that a B_i step is not an A_j step for $j \neq i$. The precise result is the following, whose proof is in the Appendix. In this theorem, letting B_i equal FALSE is equivalent to omitting that fairness condition because weak and strong fairness of FALSE are trivially true. (The action FALSE is never enabled, so (4.22) implies $\mathrm{SF}_v(\text{FALSE})$ equals $\Box\Diamond\text{FALSE} \Rightarrow \Box\Diamond\text{FALSE}$, which equals TRUE.)

Theorem 6.2 With the assumptions of Theorem 6.1, for all $i \in I$ let B_i be a subaction of A_i such that $T \wedge (i \neq j) \;\Rightarrow\; \Box[\neg(B_i \wedge A_j)]_v$ for all j in I; and let $B_i^h \triangleq \langle B_i \rangle_v \wedge (h' = \mathit{exp}_i)$. Then

$$T \,\wedge\, (\forall\, i \in I : \mathrm{XF}^i_v(B_i)) \;\;\equiv\;\; \boldsymbol{\exists}\, h : T^h \,\wedge\, (\forall\, i \in I : \mathrm{XF}^i_{vh}(B_i^h))$$

where each XF^i is either WF or SF.

6.2.3 A Completeness Result for History Variables

A popular approach to proving safety properties of concurrent programs, derived from work by Owicki and Gries [42], can prove only invariance properties. We can, in theory, reduce proving safety properties to proving invariance. We do this by adding a history variable h to a program T to obtain a program T^h. For any safety property F, we can then define a state predicate I_F that is an invariant of T^h iff (every behavior of) T satisfies F. The idea is simple: We define the value of h to be the sequence of system states in the current behavior up to and including the current state. We then define I_F to be true iff the value of h satisfies F. The result is stated in the following theorem, whose proof is sketched in the Appendix.

Theorem 6.3 Let T equal $\mathit{Init} \wedge \Box[\mathit{Next}]_{\langle \mathbf{x} \rangle}$ where $\mathbf{x}$ is the list of all variables of S; let F be a safety property such that $F(\sigma)$ depends only on the values of the variables $\mathbf{x}$ in σ, for any behavior σ; and let h be a variable not one of the variables $\mathbf{x}$. We can add h as a history variable to T to obtain T^h and define a state predicate I_F in terms of F such that $\models [\![T]\!] \Rightarrow F$ is true iff I_F is an invariant of T^h.

A simple example of the theorem is when F is the safety property F_{12} defined semantically by (4.1) of Section 4.1.2. That property asserts x must equal 1 before it can equal 2. A program $\mathit{Init} \wedge \Box[\mathit{Next}]_v$ satisfies F_{12} iff the formula $(x = 2) \Rightarrow h$ is an invariant of the program obtained by adding the history variable h to that program as follows:

$$(\mathit{Init} \wedge (h = \text{FALSE})) \;\wedge\; \Box[\mathit{Next} \wedge (h' = h \vee (x = 1))]_{v \circ \langle h \rangle}$$

Theorem 6.3 assumes only that F is a safety property. This might suggest we can show that one program satisfies the safety part of another program by verifying an invariance property. However, I have never seen this done, and in practice it seems unlikely to be possible to describe any but the simplest abstract programs with an invariant.

6.3 Stuttering Variables

Typically, when a lower-level abstract program T implements a higher-level abstract program S, program T takes more steps than S does to perform an operation. Under the refinement mapping, the extra steps of T implement stuttering steps of S. It's also possible for S to take more steps than T. In that case, defining a refinement mapping requires adding steps to behaviors of T that implement those extra steps of S. This is done by adding a *stuttering variable* s to T. The extra steps are ones that change only s, so when s is hidden, those steps become stuttering steps of T.

There are two kinds of stuttering variables used in practice: ones that add stuttering steps immediately after steps of an action, and ones that add stuttering steps immediately before steps of an action. They are described in Sections 6.3.2 and 6.3.3. Multiple such variables can be combined into a single stuttering variable. Section 6.3.5 explains another kind of stuttering variable that is never needed in practice but could, in theory, be required.

This section talks about adding stuttering steps, which literally makes no sense because it's impossible to require or forbid stuttering steps in a TLA formula. Here, adding stuttering steps to an abstract program T means writing a formula T^s containing s and the variables of T by adding steps that change s and leave the variables of T unchanged, so that $\exists\, s : T^s$ is equivalent to T. In this section, a stuttering step usually means one of those additional steps that leave the variables of T unchanged and change s.

6.3.1 The Example

Stuttering variables are explained with the silly example of a tiny censoring system. An artist paints pictures and submits them to a censor, who decides either to display or reject each picture. This system is described by the abstract program $Cen1$ defined as follows.

There are two interface variables inp and $disp$. The artist submits a picture w, which is an element of the set Art of all possible pictures, by setting the value of the variable inp to w. The censor then either displays w by setting the value of the variable $disp$ to $\langle w, i \rangle$, where i is set alternately to 0 and 1, or else rejects w. (The second component of $disp$ is needed so displaying the same picture twice isn't a stuttering step, which would needlessly complicate the example.) The censor then acknowledges receipt of the picture by setting the value of inp to a special value $NotArt$ that is not an element of Art.

There is also an internal variable aw that is hidden. The value of aw is initially the empty sequence $\langle \rangle$. It is set to $\langle w \rangle$ when the artist submits a picture w, and it is reset to $\langle \rangle$ when w is either displayed or rejected. The value of aw records whether or not the display/reject decision has been made. That information is encoded in aw this way so the example is more easily modified to obtain an example in Section 6.4. The complete description of the abstract program is formula $Cen1$ in Figure 6.4, where $ICen1$ is the program without aw hidden. (Initially, any painting may be displayed.)

There is another way to describe the artist/censor system as an abstract program. In $ICen1$, submission of a picture w by the artist is described by an input action that sets inp to w and aw to $\langle w \rangle$. A separate action $DispOrNot$ either displays or rejects w. We define $Cen2$ to equal $\exists\, aw : ICen2$ where $ICen2$ describes a program in which it is the input action that decides whether to display or reject w, setting aw to $\langle w \rangle$ iff it decides to display w. The displaying action always displays w if aw equals $\langle w \rangle$. The program $Cen2$ is defined in Figure 6.5, where v, $Init$, and Ack are the same as in $Cen1$ and are defined in Figure 6.4.

If we ignore the values of aw, the only difference between behaviors of $ICen1$ and $ICen2$ is that, when a picture is rejected, the behavior of $ICen1$ takes one more step than the corresponding behavior of $ICen2$—a step that leaves inp and $disp$ unchanged. Since inp and $disp$ are the only free variables in the two definitions of Cen, stuttering insensitivity implies that the formulas $Cen1$ and $Cen2$ are equivalent, so they describe the same abstract program.

To show that the two definitions are equivalent, we have to show that

$$
\begin{aligned}
Cen1 \;&\triangleq\; \exists\, aw : ICen1 \\
ICen1 \;&\triangleq\; Init \land \Box[Next1]_v \\
v \;&\triangleq\; \langle inp,\, disp,\, aw \rangle \\
Init \;&\triangleq\; \land\ inp = NotArt \\
&\qquad \land\ aw = \langle\, \rangle \\
&\qquad \land\ disp \in Art \times \{0,1\} \\
Next1 \;&\triangleq\; Input \lor DispOrNot \lor Ack \\
Input \;&\triangleq\; \land\ (inp = NotArt) \land (aw = \langle\, \rangle) \\
&\qquad \land\ inp' \in Art \\
&\qquad \land\ aw' = \langle inp' \rangle \\
&\qquad \land\ disp' = disp \\
DispOrNot \;&\triangleq\; \land\ aw \neq \langle\, \rangle \\
&\qquad \land\ \lor\ disp' = \langle aw(1),\, 1 - disp(2) \rangle \\
&\qquad\quad\ \ \lor\ disp' = disp \\
&\qquad \land\ aw' = \langle\, \rangle \\
&\qquad \land\ inp' = inp \\
Ack \;&\triangleq\; \land\ (inp \in Art) \land (aw = \langle\, \rangle) \\
&\qquad \land\ inp' = NotArt \\
&\qquad \land\ (aw' = aw) \land (disp' = disp)
\end{aligned}
$$

Figure 6.4: The program $Cen1$.

$$
\begin{aligned}
Cen2 \;&\triangleq\; \exists\, aw : ICen2 \\
ICen2 \;&\triangleq\; Init \land \Box[Next2]_v \\
Next2 \;&\triangleq\; InpOrNot \lor Display \lor Ack \\
InpOrNot \;&\triangleq\; \land\ (inp = NotArt) \land (aw = \langle\, \rangle) \\
&\qquad \land\ inp' \in Art \\
&\qquad \land\ \lor\ aw' = \langle inp' \rangle \\
&\qquad\quad\ \ \lor\ aw' = aw \\
&\qquad \land\ disp' = disp \\
Display \;&\triangleq\; \land\ aw \neq \langle\, \rangle \\
&\qquad \land\ disp' = \langle aw(1),\, 1 - disp(2) \rangle \\
&\qquad \land\ aw' = \langle\, \rangle \\
&\qquad \land\ inp' = inp
\end{aligned}
$$

Figure 6.5: The program $Cen2$.

$ICen1$ and $ICen2$ each implement the other under a suitable refinement mapping. We will see here how to define the refinement mapping under which $ICen2$ implements $ICen1$. Section 6.4 shows how to define the refinement mapping under which $ICen1$ implements $ICen2$.

6.3.2 Adding Stuttering Steps After an Action

To define the refinement mapping that shows $ICen2$ implements $ICen1$, we have to add a stuttering step to an execution of $ICen2$ for each operation of receiving an input and rejecting it. We do that by adding a stuttering variable that adds a stuttering step after each $InpOrNot$ step of the execution that rejects the input—that is, after $InpOrNot$ steps that set aw to $\langle\,\rangle$.

The simplest stuttering variable s is one whose value is a natural number that equals 0 except when it is adding stuttering steps (steps that change only s), in which case s equals the number of such steps it has yet to take. Here's how we add such a variable that adds stuttering steps after a subaction A of the next-state action.

Let T equal $Init \wedge \Box[Next]_v$, where $Next$ equals $A \vee (\exists j \in J : B_j)$ for actions A and B_j. We define T^s to equal $Init^s \wedge \Box[Next^s]_{vs}$, where $Next^s$ equals $A^s \vee (\exists j \in J : B_j^s)$ and $Init^s$, A^s, B_j^s, and vs are defined as follows:

S1. vs is the tuple of variables obtained by appending s to the tuple v of variables.

S2. $Init^s \;\triangleq\; Init \wedge (s = 0)$.

S3. $$A^s \;\triangleq\; \begin{array}{l} \vee\ (s = 0) \wedge A \wedge (s' = exp) \\ \vee\ (s > 0) \wedge (v' = v) \wedge (s' = s - 1) \end{array}$$
where exp is an expression whose value is a natural number; it can contain the original variables primed or unprimed.

S4. $B_j^s \;\triangleq\; (s = 0) \wedge B_j \wedge (s' = 0)$, for $j \in J$.

Ignoring the value of s, the behaviors satisfying T^s are the same as behaviors satisfying T, except each A step in a behavior of T is followed in T^s by a finite number (possibly 0) of steps that leave the variables of T unchanged. Therefore, by stuttering insensitivity, T and $\boldsymbol{\exists}\, s : T^s$ are satisfied by the same sets of behaviors, so they are equivalent.

To show that $ICen2$ implements $ICen1$, we define $ICen2^s$ in this way, where A equals $InpOrNot$ and the B_i are Ack and $Display$. In the definition of $InpOrNot^s$, we let:

$$exp \;\triangleq\; \text{IF } aw' = \langle\,\rangle \text{ THEN } 1 \text{ ELSE } 0$$

This adds a stuttering step to a behavior of $ICen2^s$ after an *InpOrNot* step that rejects the input.

Programs $ICen1$ and $ICen2^s$ take the same number of steps to process an input. A stuttering step of $ICen2^s$ corresponds to a *DispOrNot* step of $ICen1$ that rejects the input. If we compare behaviors of these two programs, we find that corresponding states have the same values of the variables *inp*, *disp*, and *aw* except when $ICen2^s$ is about to take a stuttering step. In that state, $s = 1$ in $ICen2^s$, and the value of *aw* for an input w is $\langle\,\rangle$ in $ICen2^s$ and $\langle\, w\,\rangle$ in $ICen1$. This means that the value of *aw* in a behavior of $ICen1$ always equals the value of the following state function in the corresponding behavior of $ICen2^s$:

(6.10) $awBar \;\triangleq\; \text{IF}\;\; s = 0 \;\;\text{THEN}\;\; aw \;\;\text{ELSE}\;\; \langle\, inp\,\rangle$

Therefore, $ICen2^s$ implements $ICen1$ under the refinement mapping that substitutes *awBar* for *aw*. In other words:

(6.11) $\models ICen2^s \Rightarrow (ICen1 \;\text{WITH}\; aw \leftarrow awBar)$

The proof of (6.11) is similar to, but simpler than, the refinement proof sketched in Section 5.4.2. Here, we give only the briefest outline of a proof to present results that will be used below when discussing liveness.

Let's abbreviate (F WITH $aw \leftarrow awBar$) by $\overline{F}$ for any formula F, so we must prove $\models ICen2^s \Rightarrow \overline{ICen1}$. The proof of $\models Init^s \Rightarrow \overline{Init}$ is trivial, since $s = 0$ implies $\overline{v} = v$ by definition of *awBar*, so $Init^s$ implies $\overline{Init} = Init$. The main part of the proof is proving:

(6.12) $\models ICen2^s \Rightarrow \Box\overline{[Next1]_v}$

This is proved by proving assertions C1–C4 below, which are the analogs of assertions R1–R7 of the proof in Section 5.4.2. Again, assertions containing actions of the form $\overline{\langle A\rangle_v}$ are proved for use in reasoning about liveness when a weaker assertion containing $\overline{A}$ suffices to prove (6.12). Two of the assertions require an invariant $Inv2^s$ of $ICen2^s$. That invariant needs to assert type correctness of *disp* (for C3) and that $s = 1$ implies $aw = \langle\,\rangle$ (for C2). enlargethispage

C1. $\models (s = 0) \wedge InpOrNot^s \Rightarrow \overline{Input}$

C2. $\models Inv2^s \wedge (s = 1) \wedge InpOrNot^s \Rightarrow \overline{\langle DispOrNot\rangle_v}$

C3. $\models Inv2^s \wedge Display^s \Rightarrow \overline{\langle DispOrNot\rangle_v}$

C4. $\models Ack^s \Rightarrow \overline{\langle Ack\rangle_v}$

Proving these assertions is a good way to start learning to write proofs.

Often, when showing that one program implements another, after adding a simple stuttering variable it's necessary to add a history variable to be able to define the refinement mapping. For example, suppose the input actions of the censor programs set *inp* to some value *Busy* instead of letting it be unchanged. We could then not define *awBar* to make (6.11) true because the input value would be forgotten when s equals 1. To define a state function *awBar* to make (6.11) true, we would have to add a history variable that remembers what value was input.

We can avoid having to add a history variable by letting the stuttering variable carry additional information. This is done by generalizing the way stuttering steps are counted. In the censor example, instead of setting s to 1 when adding a stuttering step and to 0 otherwise, we can set it to $\langle w \rangle$ when adding the step, where w is the value being input, and to $\langle\,\rangle$ when not adding the step. The number of stuttering steps to be taken at any point in the execution is then the length $Len(s)$ of the sequence s. We would define *awBar* to equal:

$$\text{IF } \; s = \langle\,\rangle \; \text{ THEN } \; aw \; \text{ ELSE } \; s$$

In general, we can let s assume values in any set with a well-founded relation, defined in Section 2.2.6.2 to be a set with an ordering relation in which any sequence of decreasing elements must reach a minimal element. We just require that every added stuttering step decreases the value of s.

One use of this generality is for adding stuttering steps after multiple actions. To do this, we let the value of s be a pair $\langle m, i \rangle$, where m is the number of remaining stuttering steps and i identifies the action. We define the well-founded ordering $\succ$ on this set of pairs by letting $\langle m, i \rangle \succ \langle n, j \rangle$ iff $m > n$. We can use this same trick to let the value of s be a tuple with additional components. Information in those other components can be used in defining the refinement mapping so it makes the stuttering steps implement the appropriate steps of the higher-level program. For simplicity, we state our theorem just for this particular use of a well-founded order. However, the conjunct $s(2) = i$ in the definition of A_i^s is added to ensure that only A_i^s performs stuttering steps added after A_i, although that matters only if s contains additional components that depend on i.

Theorem 6.4 (Post-Action Stuttering Variable)
Let T equal $Init \wedge \Box[Next]_v$, where $Next$ equals $(\exists\, i \in I : A_i) \vee B$ for a constant set I, and v is a tuple of all the variables of T. If T^s equals $Init^s \wedge \Box[Next^s]_{vs}$ where

- s is not a variable of T and $vs \triangleq v \circ \langle s \rangle$.

- $\mathit{Init}^s \;\triangleq\; \mathit{Init} \land (s = \langle 0, i_0 \rangle)$ for some i_0 in I.
- $\mathit{Next}^s \;\triangleq\; (\exists\, i \in I : A_i^s) \lor B^s$
- $$A_i^s \triangleq \begin{array}{l} \lor\ (s(1) = 0) \land (s(2) = i) \land A_i \land (s' = \langle \mathit{exp}_i, i \rangle) \\ \lor\ (s(1) > 0) \land (v' = v) \land (s' = \langle s(1) - 1, s(2) \rangle) \end{array}$$

  where $\models T \Rightarrow \Box[\mathit{exp}_i \in \mathbb{N}]_v$ and exp_i is a step expression not containing s.
- $B^s \triangleq (s(0) = 0) \land B \land (s' = s)$

then $\boldsymbol{\exists}\, s : T^s$ equals T.

The theorem does not assume that the actions A_i and B are mutually disjoint. A step could be both an A_i and an A_j step for $i \neq j$, or both an A_i and a B step. That should rarely be the case when applying the theorem, since it allows a nondeterministic choice of how many stuttering steps (if any) are added in some states. The action B will usually be the disjunction of actions B_j. In that case, B^s equals the disjunction of the actions $(s(0) = 0) \land B_j \land (s' = s)$.

6.3.3 Adding Stuttering Steps Before an Action

Suppose that instead of adding stuttering steps after *InpOrNot* steps of *ICen*2, we want to add them before *Ack* steps. That's a silly thing to do, but it's a silly example anyway. One thing that makes it silly is that when the *Ack* action is enabled, nothing in the state tells us whether a stuttering step is necessary. The value of *aw* is $\langle\,\rangle$ regardless of whether or not a *Display* step has occurred. So we'll have to add the stuttering step whether or not it's needed. But that's not a problem, since an unnecessary stuttering step can simply implement a stuttering step of *ICen*1.

For a simple stuttering variable that counts down to 0, we add stuttering steps before an action A the way we added them after A, except instead of A^s executing A in the first step when s equals 0, it executes A in the last step, when s equals 1 (unless it adds 0 stuttering steps). However, to ensure that an A^s step can be taken after those k stuttering steps, the stuttering steps can begin only when A is enabled. (Once A is enabled, stuttering steps leave it enabled.) To add *exp* stuttering steps before an A step, we define:

$$A^s \;\triangleq\; \begin{array}{l} \land \begin{array}{l} \lor\ \mathbb{E}(A) \land (s = 0) \land (s' = \mathit{exp}) \\ \lor\ (s > 0) \land (s' = s - 1) \end{array} \\ \land\ \textsc{if}\ \ s' = 0\ \ \textsc{then}\ \ A\ \ \textsc{else}\ \ v' = v \end{array}$$

Since A is enabled when $s' = 0$ is true, any enabling condition (conjunct with no primed variable) can be removed from A in the last line of the definition.

We could define Ack^s this way in $ICen2^s$ with $exp = 1$ to add a stuttering step before every Ack step. However, there's nothing in the state to indicate whether that stuttering step should implement a *Display* step or a stuttering step of $ICen1$. To define the refinement mapping that shows $ICen2^s$ implements $ICen1$, we would have to add a history variable that records whether or not the $InputOrNot^s$ step decided to display the input. Alternatively, we could add the history variable before adding the stuttering variable. We could then define Ack^s so it adds a stuttering step iff the *InpOrNot* step chose not to display the input.

To obtain the general result for adding stuttering steps before actions A_i, we modify Theorem 6.4 by changing the definition of A_i^s to:

$$\begin{aligned} A_i^s \triangleq\ & \land \lor\ \mathbb{E}(A_i) \land (s(1) = 0) \land (s' = \langle exp_i, i \rangle) \\ & \quad\ \lor\ (s > 0) \land (s(2) = i) \land (s' = \langle s(1) - 1, s(2) \rangle) \\ & \land \text{IF } s'(1) = 0 \text{ THEN } A_i \text{ ELSE } (v' = v) \end{aligned}$$

where exp_i is a state expression. (Although allowed, there is usually no point having primed variables in exp_i, because they equal the unprimed variables unless exp_i equals 0.)

We can also add stuttering steps both before and after A_i steps. We add a third component to s to indicate whether the next stuttering steps to be added for A_i are ones that precede or follow the A_i step. Writing a precise definition is left as an exercise for motivated readers.

6.3.4 Fairness and Stuttering Variables

As with other auxiliary variables, we add a stuttering variable to a safety property T of the form $Init \land \Box[Next]_v$. If a program is described by the property $T \land L$ for a liveness property L, then the program with the added stuttering variable s is $T^s \land L$.

To see how stuttering variables work with liveness conditions, we add fairness requirements $L1$ and $L2$ to our two censor programs to define:

$$IC1 \triangleq ICen1 \land L1 \qquad IC2 \triangleq ICen2 \land L2$$

We've shown that $ICen2^s$ implements $ICen1$ under a refinement mapping. We show here that $IC2^s$ implements $IC1$ under that same refinement mapping. That is, we show:

(6.13) $\models IC2^s \Rightarrow (IC1 \text{ WITH } aw \leftarrow awBar)$

The fairness requirements are:

$$\begin{aligned} L1 &\triangleq \mathrm{WF}_v(\mathit{DispOrNot}) \,\wedge\, \mathrm{WF}_v(\mathit{Ack}) \\ L2 &\triangleq \mathrm{WF}_v(\mathit{Display}) \,\wedge\, \mathrm{WF}_v(\mathit{Ack}) \end{aligned}$$

(Theorem 4.7 implies that $L1$ and $L2$ are equivalent to weak fairness of $\mathit{DispOrNot} \vee \mathit{Ack}$ and $\mathit{Display} \vee \mathit{Ack}$ respectively, but it's more convenient to write them this way.) It's clear that $L1$ and $L2$ are the appropriate fairness requirements for $IC1$ and $IC2$, ensuring that an Ack step occurs after each input step. In particular, an input step of $IC2$ is a $\mathit{DispOrNot}$ step, after which eventually Ack is enabled—either immediately if the input is rejected or after a $\mathit{Display}$ step that $\mathrm{WF}_v(\mathit{Display})$ implies must occur. When Ack is enabled, $\mathrm{WF}_v(\mathit{Ack})$ implies that an Ack step must occur.

For $IC2^s$ to implement $IC1$ under a refinement mapping, it should ensure that an input step is eventually followed by an Ack^s step. In $IC2^s$, an input is entered by an $(s = 0) \wedge \mathit{InpOrNot}^s$ step. We must show that such a step is eventually followed by an Ack^s step. This appears problematic because if the step rejects the input, then it sets s to 1, in which case the only enabled action of $\mathit{Next}2^s$ is $(s = 1) \wedge \mathit{InpOrNot}^s$; and $L2$ asserts no fairness condition for that action. To show that the $(s = 0) \wedge \mathit{InpOrNot}^s$ step must be followed by an Ack^s step, we first show as follows that $\exists\, s : IC2^s$ is equivalent to $IC2$:

$$\begin{aligned} \exists\, s : IC2^s &\equiv \exists\, s : \mathit{ICen}2 \,\wedge\, L2 && \text{By definition of } IC2^s. \\ &\equiv (\exists\, s : \mathit{ICen}2^s) \,\wedge\, L2 && \text{Because } s \text{ does not occur in } L2. \\ &\equiv \mathit{ICen}2 \,\wedge\, L2 && \text{By Theorem 6.4.} \\ &\equiv IC^2 && \text{By definition of } IC2^s. \end{aligned}$$

Any behavior that satisfies $IC2^s$ satisfies $\exists\, s : IC2^s$, so it satisfies $IC2$. An $(s = 0) \wedge \mathit{InpOrNot}^s$ step is an $\mathit{InpOrNot}$ step, which by $IC2$ must eventually be followed by an Ack step, which by definition of $\mathit{ICen}2^s$ must be an Ack^s step. Thus, $IC2^s$ implies that any input step is eventually followed by an Ack^s step.

In the case of the input being rejected, the necessary $(s = 1) \wedge \mathit{InpOrNot}^s$ step must occur to satisfy the fairness requirement $\mathrm{WF}_v(\mathit{Ack})$ on the action Ack in $IC2$. If you think of the abstract program $IC2^s$ as instructions to a computer for generating behaviors, then this makes no sense. How can a fairness condition on Ack tell the computer to take an $\mathit{InpOrNot}^s$ step? But by now, you should understand that an abstract program is a predicate on behaviors, not instructions for generating them. Formula $\mathit{ICen}2^s \wedge \mathrm{WF}_v(\mathit{Ack})$ implies that if a state with $s = 1$ has been reached, then there must be another $\mathit{InpOrNot}^s$ step and then an Ack^s step in the behavior.

This may seem weird. The source of the apparent weirdness is that ICS^2 contains a fairness condition on the action Ack, which is not a subaction of the next-state action $\mathit{Next}2^s$. Fairness conditions on actions not a subaction of the next-state action can lead to weirdness, including program descriptions that are not machine closed. However, in this case, we still get a machine-closed program description. In fact, this is true in general. If $\langle T, L\rangle$ is machine closed and T^s is obtained from T by adding a stuttering variable, then $\langle T^s, L\rangle$ is also machine closed. The proof is the same as the one for history variables sketched in Section 6.2.2, except in defining the behavior τ, we may have to add stuttering steps to σ as well as changing the values of the variable s. Stuttering insensitivity of L implies that $\rho \circ \tau$ still satisfies L.

We now explain the proof of (6.13). As before, define $\overline{F}$ to equal (F WITH $aw \leftarrow \mathit{awBar}$) for any formula F, so (6.13) asserts $\models \mathit{IC}2^s \Rightarrow \overline{\mathit{IC}1}$. The proof of $\models \mathit{ICen}2^s \Rightarrow \overline{\mathit{ICen}1}$ is discussed in Section 6.3.2, so we consider only the proof of $\models \mathit{IC}2^s \Rightarrow \overline{L1}$, which requires proving:

(6.14) (a) $\models \mathit{IC}2^s \;\Rightarrow\; \overline{\mathrm{WF}_v(\mathit{DispOrNot})}$
(b) $\models \mathit{IC}2^s \;\Rightarrow\; \overline{\mathrm{WF}_v(\mathit{Ack})}$

We now sketch a proof of (6.14a). As usual when proving temporal properties, instead of assuming $\mathit{IC}2^s$, which is true only for a behavior starting in a state satisfying $\mathit{Init}2^s$, we assume this $\Box$ formula implied by $\mathit{IC}2^s$

$$\mathit{IIC}2^s \;\triangleq\; \Box \mathit{Inv}2 \,\wedge\, \Box[\mathit{Next}2^s]_{vs} \,\wedge\, L2$$

where $\mathit{Inv}2$ is an invariant of $\mathit{ICen}2^2$ that asserts some obvious invariants such as type correctness. Here is the proof sketch.

1. SUFFICES: ASSUME: $\mathit{IIC}2^s \wedge \Box\overline{\mathbb{E}\langle \mathit{DispOrNot}\rangle_v} \wedge \Box\overline{[\neg \mathit{DispOrNot}]_v}$
PROVE: $\Diamond\overline{\langle \mathit{DispOrNot}\rangle_v}$

PROOF: By (4.15).

2. $\Box((aw \neq \langle\,\rangle) \vee (s \neq 0))$

PROOF: The definition of $\mathit{DispOrNot}$ implies that $\overline{\mathbb{E}\langle \mathit{DispOrNot}\rangle_v}$ equals $\overline{aw \neq \langle\,\rangle}$, so the step 1 assumption $\Box\overline{\mathbb{E}\langle \mathit{DispOrNot}\rangle_v}$ implies $\Box\overline{aw \neq \langle\,\rangle}$; and the definition of awBar implies $\overline{aw \neq \langle\,\rangle}$ equals $(aw \neq \langle\,\rangle) \vee (s \neq 0)$.

3. $\Diamond\Box(aw \neq \langle\,\rangle) \;\vee\; \Diamond\Box(s \neq 0)$

3.1. $\Box\,(\,(aw \neq \langle\,\rangle) \;\Rightarrow\; \Box(aw \neq \langle\,\rangle)\,)$

PROOF: The assumption $\overline{IIC2^s}$ implies that $aw \neq \langle\,\rangle$ can be made false only by a $Display^s$ step, which by C3 is a $\overline{\langle DispOrNot\rangle_v}$ step. The assumption $\Box\overline{[\neg DispOrNot]_v}$ implies that such a step can't occur. Therefore, if $aw \neq \langle\,\rangle$ ever becomes true, then it must remain true forever.

3.2. Q.E.D.

PROOF: By steps 2 and 3.1 and the temporal logic tautology:

$$\models \Box(F \vee G) \wedge \Box(F \Rightarrow \Box F) \;\Rightarrow\; (\Diamond\Box F \vee \Diamond\Box G)$$

4. CASE: $\Diamond\Box(aw \neq \langle\,\rangle)$

PROOF: Since $aw \neq \langle\,\rangle$ equals $\mathbb{E}\langle Display\rangle_v$, the case assumption and $\mathrm{WF}_v(Display)$ imply that, when $\Box(aw \neq \langle\,\rangle)$ becomes true, a $\langle Display\rangle_v$ step eventually occurs, and $\overline{IIS^s}$ implies that this step must be a $Display^s$ step. By C3, this $Display^s$ step is a $\overline{\langle DispOrNot\rangle_v}$ step, which implies the goal introduced by step 1.

5. CASE: $\Box(s \neq 0)$

PROOF: The case assumption and the assumption $\Box Inv2$ imply $\Box(s = 1)$. As shown above in the explanation of why a behavior of $IC2^s$ can't halt in a state with $s = 1$, the property $\mathrm{WF}_v(Ack)$ implies that, in such a state, an $(s = 1) \wedge InpOrNot^s$ step must eventually occur. By C2, that is the $\overline{\langle DispOrNot\rangle_v}$ step that proves the step 1 goal.

6. Q.E.D.

PROOF: Step 3 implies that the step 4 and 5 cases are exhaustive.

The proof of (6.14b) is similar but simpler, since it doesn't have the complication of deducing from fairness of one action (Ack^s) that a step of another action ($DispOrNot^s$) of the same program must occur.

Theorem 6.2 shows how, after adding a history variable to a program, we can rewrite the program's fairness properties as fairness conditions of subactions of the modified program's next-state action. I don't know if there is a similar result for stuttering variables. Theorem 6.2 is relevant to methods other than TLA for describing abstract programs. Those other methods that I'm aware of do not assume stuttering insensitivity, so a similar result for stuttering variables seems to be of no interest.

6.3.5 Infinite-Stuttering Variables

Suppose a terminating program is described by a formula $\mathbf{\exists}\, y : IS$, where IS implies that the value of y keeps changing forever. (IS implies that at some point, the values of all its other variables stop changing.) Suppose also that program $\mathbf{\exists}\, y : IS$ is refined by a terminating program T with no internal variables, so all its variables eventually stop changing. The methods of adding stuttering steps to a program described so far add a finite number of stuttering steps to non-stuttering steps of the program. They can't define a state function that keeps changing forever, so they can't be used to define a refinement mapping to show that T implements $\mathbf{\exists}\, y : IS$.

It's easy to construct an example of such programs IS and T, but I can't imagine one occurring in practice. We consider them only for completeness—and in particular, to prove the completeness theorem in Section 6.4.4 stating that if $\models T \Rightarrow \mathbf{\exists}\, \mathbf{y} : IS$ is true for some tuple $\mathbf{y}$ of variables, then we can add auxiliary variables to T to obtain a program T^a that implements IS under a refinement mapping. For that theorem to be true, we need to define an *infinite-stuttering* variable whose value keeps changing forever to handle this situation that never occurs in practice.

There are lots of ways to define an infinite-stuttering variable. Here is the definition used in the proof of Theorem 6.6. Let T equal $Init \wedge \Box[Next]_v$, where v is the tuple of all variables that appear in T, and let s not be one of those variables. We then define T^s to equal:

$$Init \wedge \Box[(Next \wedge (v' \neq v)) \vee ((s' \neq s) \wedge (v' = v)]_{v \circ \langle s \rangle} \wedge \Box\Diamond\langle s' \neq s\rangle_s$$

6.4 Prophecy Variables

6.4.1 Simple Prophecy Variables

We observed in Section 6.3.1 that the descriptions $Cen1$ and $Cen2$ of the censor system were equivalent. We showed that $Cen2$ implies $Cen1$, which required adding a stuttering variable to $ICen2$. We now complete the demonstration of equivalence by showing that $Cen1$ implies $Cen2$. This requires defining a state function $awBar$ such that $ICen1$ implies:

$$(ICen2 \text{ WITH } aw \leftarrow awBar)$$

However, this is impossible for the following reason. Because the refinement mapping substitutes the variables *inp* and *disp* of $ICen1$ for the corresponding variables of $ICen2$, an *Input* step of $ICen1$ must implement an *InpOrNot* step of *ICen*2. Besides choosing the input, the *InpOrNot* action of *ICen*2 also decides whether or not that input is to be displayed, recording its decision in the value of *aw*. However, that decision is made by *ICen*1 later, when executing the *DispOrNot* action. Immediately after the *Input* action, there's no information in the state of *ICen*1 to determine what the value of variable *aw* of *ICen*2 should be.

The solution to this problem is to have the *Input* action guess what *DispOrNot* will do, indicating its guess by setting a *prophecy variable p* to a value that predicts whether the input will be displayed or rejected by the *DispOrNot* step.

To make the generalization from this example more obvious, let's write action *DispOrNot* of *ICen*1 as the disjunction of two actions: $DorN_{Yes}$ that displays the input and $DorN_{No}$ that doesn't. Remember that:

$$\begin{array}{lll} DispOrNot & \triangleq & \land \ \ldots \\ & & \land \lor \ disp' = \langle aw(1), 1 - disp(2) \rangle \\ & & \quad \lor \ disp' = disp \\ & & \vdots \end{array}$$

We can define $DorN_i$, for $i = Yes$ and $i = No$, by modifying the definition of *DispOrNot* to get:

$$\begin{array}{lll} DorN_i & \triangleq & \land \ \ldots \\ & & \land \lor (i = Yes) \land (disp' = \langle aw(1), 1 - disp(2) \rangle) \\ & & \quad \lor (i = No) \ \ \land (disp' = disp) \\ & & \vdots \end{array}$$

We then replace *DispOrNot* in *ICen*2 by $\exists\, i \in \Pi : DorN_i$, where Π equals $\{Yes, No\}$. We can then add to *ICen*2 an auxiliary variable *p* called a prophecy variable to obtain a formula $ICen2^p$ in which the *Input* action is replaced by

$$Input^p \ \triangleq \ Input \land (p' \in \Pi)$$

and the *DispOrNot* action is replaced by:

$$DispOrNot^p \ \triangleq \ DorN_p$$

Thus the $Input^p$ action predicts what the *DispOrNot* action will do, and $DispOrNot^p$ is modified to ensure that the prediction comes true. To complete the definition of $ICen1^p$, we can let $Init^p$ equal *Init* and Ack^p equal *Ack*, since the value of *p* matters only after an $Input^p$ step and before the following $DispOrNot^p$ step.

In $ICen2$, the value of aw is $\langle\,\rangle$ except after an $InpOrNot$ step that chose to display the input. This implies

(6.15) $\models ICen1^p \Rightarrow (ICen2 \text{ WITH } aw \leftarrow awBar)$

where $awBar$ is defined by:

$$awBar \;\triangleq\; \text{IF } aw \neq \langle\,\rangle \wedge (p = Yes) \text{ THEN } aw \text{ ELSE } \langle\,\rangle$$

To show that (6.15) implies $\models ICen1 \Rightarrow ICen2$, we have to show that p is an auxiliary variable—that is, we have to show that $\exists\, p : ICen1^p$ is equivalent to $ICen1$. To do that, we prove two things:

1. Every behavior satisfying $ICen1^p$ satisfies $ICen1$

   PROOF: It's clear that $Init^p$ equals $Init$, Act^p equals Act, and $Input^p$ implies $Input$. To complete the proof, we must show that every $DispOrNot^p$ step is a $DispOrNot$ step. It's easy to see that

   $$\mathbb{E}(DispOrNot^p) \;\Rightarrow\; (p \in \{Yes, No\})$$

   is an invariant of $ICen1^p$, and to check that $DispOrNot^p$ implies $DispOrNot$ for each of those two values of p.

2. For any behavior σ satisfying $ICen1$ there is a behavior τ satisfying $ICen1^p$ such that $\sigma \simeq_p \tau$.

   PROOF: We let σ be a behavior satisfying $ICen1$ and construct the states of τ from the states of σ by specifying the value of p in each of those states, so obviously $\sigma \simeq_p \tau$. The behavior τ will satisfy $ICen1^p$ if the values chosen for p satisfy these three conditions:

   1. The value of p in the second state of a stuttering step of σ (one leaving the variables of $ICen1$ unchanged) is the same as its value in the first state of the step.
   2. After an $Input^p$ step, the value of p is either Yes or No.
   3. In the first state of a $DispOrNot$ step of σ, the value of p must make that step a $DispOrNot^p$ step of τ.

   We define the values of p in all states of τ as follows. We let p have any value in the initial state. In any other state of τ, we let the value of p be the same as its value in the previous state except if the state is the second state of an $Input$ step of σ. In that case, we let the value of p equal Yes if the next $DispOrNot$ step of σ changes $disp$; otherwise we let it equal No. (If there is no next $DispOrNot$ step, so there remain only stuttering steps, we can let p have either value.) It's easy to check that this way of defining p makes it satisfy the three conditions. END PROOF

Let's now generalize from this example. We want an action B to predict the result of the next execution of an action A. We do this by writing A as $\exists\, i \in \Pi : A_i$ for a constant set Π of possible predictions and having B predict for which value of i the next A step will be an A_i step. Action B makes the prediction by setting the variable p to equal its prediction, so we define B^p to equal $B \land (p' \in \Pi)$. The prediction is made to come true by defining A^p to equal A_p.

One way our example was special is that the prediction made by an Input^p step is used by $\mathit{DispOrNot}^p$ in the first non-stuttering step after it is made. This allowed $\mathit{ICen}1^p$ to leave the new value of p unspecified by other actions. Usually, there can be steps of other actions between when the prediction is made and when it is fulfilled. Those other actions should leave the value of p unchanged. For any subaction B of the next-state action other than A, we let B^p equal either $B \land (p' = p)$ if it doesn't make a prediction or $B \land (p' \in \Pi)$ if it makes one. It doesn't matter if multiple predictions are made for the same A step; only the most recent one counts.

There is no reason not to let p always equal an element of Π, so we will do that. This means we let Init^p equal $\mathit{Init} \land (p \in \Pi)$. This can represent an initial prediction, or it can be overridden by a subsequent prediction. In either case, it means that we have the type invariant $\Box(p \in \Pi)$.

It doesn't matter if a prediction is never used—either because it is overridden by another prediction or an A step never occurs. What does matter is that a prediction must be used at most once. Our ability to choose the right prediction in the proof that $\boldsymbol{\exists}\, p : \mathit{ICen}1^p$ is equivalent to $\mathit{ICen}1$ depended on this. To make sure that this is true, we require that A^p makes a prediction, so we define A^p to equal $A_p \land (p' \in \Pi)$. That prediction can always be overridden by a subsequent prediction made by a different action. The argument above that the variable p in our example was a prophecy variable then generalizes to prove:

Theorem 6.5 (Simple Prophecy Variable) Let $T \triangleq \mathit{Init} \land \Box[\mathit{Next}]_v$ where v is the tuple of variables in T, let

$$\mathit{Next} \;\triangleq\; (\exists\, i \in \Pi \,:\, A_i) \lor (\exists\, j \in J \,:\, B_j)$$

and let Π be a constant set. If p is not a variable of T,

$$\begin{aligned} T^p &\triangleq \mathit{Init}^p \land \Box[\mathit{Next}^p]_{vp}\,, \\ vp &\triangleq v \circ \langle p \rangle\,, \\ \mathit{Init}^p &\triangleq \mathit{Init} \land (p \in \Pi)\,, \\ \mathit{Next}^p &\triangleq (A_p \land (p' \in \Pi)) \lor (\exists\, j \in J \,:\, B_j \land C_j)\,, \end{aligned}$$

and each C_j equals $p' = p$ or $p' \in \Pi$, then $\models (\boldsymbol{\exists}\, p : T^p) \equiv T$.

The theorem makes no disjointness assumption about the actions A_i and B_j, but in most applications of the theorem they will be mutually disjoint.

It is inelegant and possibly confusing to have a program make predications that are never used—for example, by having the $\mathit{DispOrNot}^p$ action of $\mathit{ICens}1^p$ make a prediction that is always replaced by the prediction made by the Input^p action. If the prediction will never be used, we can replace $p' \in \Pi$ by $p' = \mathit{None}$ (or $p \in \Pi$ by $p = \mathit{None}$ for an initial prediction), where None is a value not in Π. The assertion that the prediction is never used means that the following state predicate is an invariant of T^p:

$$(p = \mathit{None}) \Rightarrow \neg\, \mathbb{E}(\exists\, i \in \Pi : A_i)$$

We could also modify Next^p to allow a special value of p indicate that no prediction is being made, but there is no reason to do that.

6.4.2 Predicting the Impossible and Liveness

What if a prophecy variable makes a prediction that can't be fulfilled? A prophecy variable predicts, for an action A equal to $\exists\, i \in \Pi : A_i$, the value of i for which the next A step is an A_i step. The prediction that the next A step will be an A_p step can't be fulfilled if action A_p can't be enabled until an A_j step occurs for some $j \neq p$.

Let's look at the worst case: a prediction that predicts that the next A step will be an A_i step, where A_i equals FALSE. We can write any next-state action Next as

$$\exists\, i \in \{0,1\} : ((i = 1) \land \mathit{Next}) \lor ((i = 0) \land \text{FALSE})$$

(If $i = 0$, then a $[\mathit{Next}]_v$ step leaves the variables of v unchanged.) The observation that $\exists\, i \in \{0,1\} : F_i$ equals $F_0 \lor F_1$ and a bit of propositional logic show that

$$\models \exists\, i \in \{0,1\} : (p = i) \land (((i = 1) \land \mathit{Next}) \lor ((i = 0) \land \text{FALSE}))$$

equals $(p = 1) \land \mathit{Next}$. Theorem 6.5 therefore implies that if T equals $\mathit{Init} \land \Box[\mathit{Next}]_v$, then T equals $\boldsymbol{\exists}\, p : T^p$ where

$$\begin{aligned} T^p \;\triangleq\; & \land\ \mathit{Init} \land (p \in \{0,1\}) \\ & \land\ \Box\,[\,(p = 1) \land \mathit{Next} \land (p' \in \{0,1\})\,]_{vp} \end{aligned}$$

In other words, if p ever becomes equal to 0, then the next-state relation of T^p is never again enabled, so the behavior halts with an infinite sequence of stuttering steps—ones that leave p and the variables of T unchanged.

But that's perfectly OK. T is a safety property; it allows behaviors that terminate at any point. The prophecy variable p is simply predicting whether the behavior will terminate before the next *Next* step.

If we are describing an abstract program in which $\langle \mathit{Next} \rangle_v$ is always enabled and its execution is never supposed to stop, then we must conjoin to T some fairness property, such as $\mathrm{WF}_v(\mathit{Next})$. If $\langle \mathit{Next} \rangle_v$ is enabled in every reachable state of T, then it is enabled in every reachable state of T^p, since the reachable states of T^p are reachable states of T because T equals $\exists\, p : T^p$. In that case, conjoining $\mathrm{WF}_v(\mathit{Next})$ to T^p adds the requirement that in every behavior, an infinite number of non-stuttering *Next* steps must occur. In our worst-case example, $p = 0$ implies $\mathit{Next}^p =$ FALSE, so $T^p \wedge \mathrm{WF}_v(\mathit{Next})$ is satisfied only by behaviors in which infinitely many $\langle \mathit{Next} \rangle_v$ steps occur, and hence in which p never equals 0.

Conjoining $\mathrm{WF}_v(\mathit{Next})$ to T^p rules out finite behaviors allowed by T^p—ones in which p equals 0. Hence, the pair $\langle T^p, \mathrm{WF}_v(\mathit{Next}) \rangle$ is not machine closed, so $\mathrm{WF}_v(\mathit{Next})$ is not a fairness property for T^p. This doesn't contradict Theorem 4.6, because *Next*, which is a trivial subaction of *Next*, is not a subaction of Next^p. In general, if predicting that the next A step is an A_p step is a nontrivial predication, then every possible A step can't be a Next^p step, so $\models A \Rightarrow \mathit{Next}^p$ can't be true—which by definition means A is not a subaction of Next^p.

As dramatically illustrated by this example, adding a prophecy variable that can make impossible predictions to a description of an abstract program with a fairness property produces a ⟨safety, liveness⟩ pair that is not machine closed. Although this is not a typical example, in practice prophecy variables often do make impossible predictions. This is usually because it's easier not to eliminate them. That's the case for the example in Section 6.5.

Programs that are not machine closed are weird, and unintentional weirdness usually indicates an error. An abstract program that describes how a concrete program works should be machine closed, because coding languages have no way of expressing liveness properties that are not fairness properties. Abstract programs that are not machine closed should almost always be avoided because they're hard to understand. However, there are exceptions [25, Section 3.2]. On the other hand, prophecy variables are added to a program only for verifying that it implements another program. There is no reason adding a prophecy variable should produce a machine closed program.

6.4.3 General Prophecy Variables

A simple prophecy variable makes a single prediction. General prophecy variables can make multiple predictions. Those multiple predictions can be successive predictions about a single action or separate predictions about different actions. These two possibilities are illustrated with variants of the censor system. There can also be multiple predictions about multiple actions, but we won't try to be that general. The two examples illustrate the concepts. A very general definition has been described elsewhere for expert TLA$^+$ users [37].

6.4.3.1 A Sequence of Prophecies

Let's now modify the censor programs to allow the artist to submit a new picture before the censor has either displayed or rejected the previous submission. At any time, there may be a queue of submissions being processed by the censor. In the modified version of *ICen*1, called *ICenSeq*1, the censor has not yet decided whether to display or reject any of the submissions in that queue. In *ICenSeq*2, the modified version of *ICen*2, the censor decides immediately whether to accept or reject a submission and maintains only a queue of submissions to be displayed.

Formula *CenSeq*1 is defined in Figure 6.6, where everything is in gray except for parts that differ from the corresponding parts of the definition of *Cen*1 in Figure 6.4 other than by adding "*Seq*" to names. Because of the way we defined *Cen*1, with *aw* equal to a sequence of 0 or 1 pictures, the changes are minimal. (Recall the definitions of *Tail* and *Append* from Section 2.3.2.)

Similarly, Figure 6.7 shows the definition of *CenSeq*2, using formulas defined in Figure 6.6. Shown in black are the parts that differ from the corresponding parts in the definition of *Cen*2 in Figure 6.5 by more than a name change.

In both *ICenSeq*1 and *ICenSeq*2, the value of the variable *aw* is the queue being maintained by the censor. As with *ICen*1 and *ICen*2, when *aw* is hidden by $\exists$, the two formulas are equivalent. As in the previous example, *ICenSeq*2 decides whether to display or reject an input before *ICenSeq*1 does. To define a refinement mapping to show *ICenSeq*1 implements *ICenSeq*2, we need to add a prophecy variable p to *ICenSeq*1 that is set by the *Input* action and predicts the decisions that will be made by the *DispOrNotSeq* action. However, this time there are multiple predictions to be remembered—one for every picture in *aw*.

$$CenSeq1 \triangleq \exists\!\!\exists\, aw : ICenSeq1$$

$$ICenSeq1 \triangleq Init \land \Box[NextSeq1]_v$$

$$v \triangleq \langle inp, disp, aw \rangle$$

$$InitSeq \triangleq \begin{array}{l} \land\ inp = NotArt \\ \land\ aw = \langle\,\rangle \\ \land\ disp \in Art \times \{0, 1\} \end{array}$$

$$NextSeq1 \triangleq InputSeq \lor DispOrNotSeq \lor AckSeq$$

$$InputSeq \triangleq \begin{array}{l} \land\ inp = NotArt \\ \land\ inp' \in Art \\ \land\ aw' = Append(aw, inp') \\ \land\ disp' = disp \end{array}$$

$$DispOrNotSeq \triangleq \begin{array}{l} \land\ aw \neq \langle\,\rangle \\ \land\ \begin{array}{l} \lor\ disp' = \langle aw[1], 1 - disp(2) \rangle \\ \lor\ disp' = disp \end{array} \\ \land\ aw' = Tail(aw) \\ \land\ inp' = inp \end{array}$$

$$AckSeq \triangleq \begin{array}{l} \land\ inp \in Art \\ \land\ inp' = NotArt \\ \land\ (aw' = aw) \land (disp' = disp) \end{array}$$

Figure 6.6: The program *CenSeq*1.

$$CenSeq2 \triangleq \exists\!\!\exists\, aw : ICenSeq2$$

$$ICenSeq2 \triangleq InitSeq \land \Box[NextSeq2]_v$$

$$NextSeq2 \triangleq InpOrNotSeq \lor DisplaySeq \lor AckSeq$$

$$InpOrNotSeq \triangleq \begin{array}{l} \land\ inp = NotArt \\ \land\ inp' \in Art \\ \land\ \begin{array}{l} \lor\ aw' = Append(aw, inp') \\ \lor\ aw' = aw \end{array} \\ \land\ disp' = disp \end{array}$$

$$DisplaySeq \triangleq \begin{array}{l} \land\ aw \neq \langle\,\rangle \\ \land\ disp' = \langle aw[1], 1 - disp(2) \rangle \\ \land\ aw' = Tail(aw) \\ \land\ inp' = inp \end{array}$$

Figure 6.7: The program *CenSeq*2.

You have probably figured out that this will be done by letting the value of p be a sequence of *Yes* or *No* values, each element of p predicting whether the corresponding input in the sequence aw will be displayed or rejected by the *DispOrNot* action. Here's how we define $ICenSeq1^p$, the formula obtained by adding the prophecy sequence variable p to $ICenSeq1$.

Let Π be the set $\{\mathit{Yes}, \mathit{No}\}$ of predications. The value of p should always be a sequence of elements of Π having the same length as the value of the variable aw of $ICenSeq1$. The initial predicate of $ICenSeq1^p$ is:

$$Init^p \;\triangleq\; InitSeq \land (p = \langle\,\rangle)$$

In addition to appending the input to aw, the action $InputSeq^p$ must append to p the prediction of whether or not that input will be displayed:

$$InputSeq^p \;\triangleq\; InputSeq \;\land\; (\exists\, i \in \Pi \,:\, p' = Append(p, i))$$

As in $ICen1^p$, to make $DispOrNotSeq^p$ display the input iff p predicts that it will, we define $DorNSeq_i$ so that

$$DispOrNotSeq \;\triangleq\; \exists\, i \in \Pi \,:\, DorNSeq_i$$

where $DorNSeq_{Yes}$ displays the input and $DorNSeq_{No}$ rejects it. The definition of $DorNSeq_i$ is obtained by modifying $DispOrNotSeq$ the same way we modified $DispOrNot$ to obtain $DorN_i$ for $ICen1$. We can then define:

$$DispOrNotSeq^p \;\triangleq\; DorNSeq_{p(1)} \land (p' = Tail(p))$$

Note that having $DispOrNotSeq^p$ set p' to $Tail(p)$ ensures that every prediction is used only once. Since $AckSeq^p$ neither makes nor satisfies a prediction, we define:

$$AckSeq^p \;\triangleq\; AckSeq \;\land\; (p' = p)$$

Putting this all together we get:

$$ICenSeq1^p \;\triangleq\; InitSeq^p \;\land\; \Box[NextSeq1^p]_{vp}$$

where

$$NextSeq1^p \;\triangleq\; InputSeq^p \;\lor\; DispOrNotSeq^p \;\lor\; AckSeq^p$$

and vp equals $\langle\, inp, disp, aw, p\,\rangle$.

We can now show that $CenSeq1$ implements $CenSeq2$ by showing

$$(6.16)\quad \models ICenSeq1 \;\Rightarrow\; (ICenSeq2 \text{ WITH } aw \leftarrow awBar)$$

where *awBar* is the subsequence of *aw* containing only the pictures that p predicts will be displayed. To define *awBar*, we first define $OnlyYes(wsq, ysq)$ to be the subsequence of the sequence *wsq* consisting of all elements for which the corresponding elements of the sequence *ysq* equals *Yes*. The definition assumes that *wsq* and *ysq* are sequences with the same length. Here is the recursive definition. (Recall that "$\circ$" is concatenation of sequences.)

```
$OnlyYes(wsq, ysq) \triangleq$
    IF $wsq = \langle\,\rangle$ THEN $\langle\,\rangle$
                    ELSE ( IF $Head(ysq) = Yes$ THEN $\langle Head(wsq)\rangle$
                                                ELSE $\langle\,\rangle$ )
                           $\circ\ OnlyYes(Tail(wsq), Tail(ysq))$
```

Defining *awBar* to equal $OnlyYes(aw, p)$ makes (6.16) true.

It's straightforward to modify Theorem 6.5 to describe an arbitrary prophecy variable p that makes a sequence of predictions. We replace the definition of $Next^p$ in the hypothesis of the theorem by:

$$Next^p \triangleq (A_{p(1)} \wedge D) \vee (\exists j \in J : B_j \wedge C_j), \text{ where}$$
$$D \text{ equals } p' = Tail(p) \text{ or } \exists i \in \Pi : p' = Append(Tail(p), i)$$
$$C_j \text{ equals } p' = p \text{ or } \exists i \in \Pi : p' = Append(p, i)$$

However, there's one problem: The empty sequence $\langle\,\rangle$ is the value of p indicating that no prediction is being made. When $p = \langle\,\rangle$, the value of the subscript $p(1) = i$ in this definition is undefined. That doesn't matter in our example because p and *aw* are sequences of the same length, so $p = \langle\,\rangle$ implies $aw = \langle\,\rangle$, which implies that $DorNSeq_i$ equals FALSE for $i \in \Pi$. Therefore, the value of the undefined subformula makes no difference. In general, to make the modified theorem valid, we need to add to its hypothesis the requirement that the following is an invariant of T^p:

$$(p = \langle\,\rangle) \Rightarrow \neg\, \mathbb{E}\,(\exists i \in \Pi : A_i)$$

6.4.3.2 A Set of Prophecies

To illustrate a prophecy variable that makes a set of concurrent predictions, we now modify the censor programs *CenSeq*1 and *CenSeq*2 so that instead of displaying pictures in the order in which they were submitted, the censor can display them in any order. This is represented by letting *aw* be a set rather than a sequence of pictures. It is done in the two programs *ICenSet*1 and *ICenSet*2, where the first lets *aw* be the set of all unprocessed inputs and the second lets *aw* contain just the ones that will be displayed. Letting

*CenSet*1 and *CenSet*2 be the programs obtained from these two programs by hiding *aw*, we want to show that *CenSet*1 implements *CenSet*2. As you probably realize, defining a refinement mapping to show that this is true requires adding a prophecy variable p to *ICenSet*1 that predicts which of the inputs in *aw* will be displayed.

Writing these two censor programs poses a problem. What if the artist submits the same picture twice? If we want the picture to be displayed twice, we would need to have two copies of it in *aw*, which means *aw* couldn't simply be a set. In the example of Section 6.5, you'll see one way of keeping multiple copies of a value in a set. But for simplicity, we'll modify the censor programs not to allow the artist to submit the same picture twice. This will be done by adding an interface variable *old* whose value is the set of all previously submitted pictures.

The definition of *CenSet*1 is in Figure 6.8, with the changes from the definition of *CenSeq*1 (Figure 6.6) in black. You should be able to write the definition of *CenSeq*2 yourself.

To define a refinement mapping under which *ICenSet*1 implements *ICenSet*2, we need to add a prophecy variable p to *ICenSet*1 that predicts, for each picture in *aw*, whether or not that picture will be displayed. The obvious way to do that is to let the value of p be a function in $aw \rightarrow \Pi$, the set of functions from *aw* to Π. As before, we let Π equal the set $\{Yes, No\}$.

Since *aw* initially equals the empty set, the initial value of p should be the function whose domain is the empty set. There is just a single such function, and the easiest way to write it is as the empty sequence $\langle\,\rangle$, which is a (and hence the) function whose domain is the empty set. So, we define:

$$InitSet^p \;\triangleq\; InitSet \wedge (p = \langle\,\rangle)$$

The $InputSet^p$ action must add a prediction of whether or not the picture inp' that it adds to *aw* will be displayed. Thus, it must assert that p' is the function obtained from p by adding inp' to its domain and letting the value of $p'(inp')$ be either element in Π. To write that action, let's define $FcnPlus(f, w, d)$ to be the function obtained from a function f by adding an element w to its domain and letting that function map w to d. The domain of f is written $\textsc{domain}(f)$, so the definition is:

$$\begin{array}{l} FcnPlus(f, w, d) \;\triangleq \\ \quad x \in \{w\} \cup \textsc{domain}(f) \;\mapsto\; \textsc{if}\;\; x = w \;\;\textsc{then}\;\; d \;\;\textsc{else}\;\; f(x) \end{array}$$

We can then define

$$InputSet^p \;\triangleq\; InputSet \wedge (\exists\, i \in \Pi : p' = FcnPlus(p, inp', i))$$

$$
\begin{array}{l}
\mathit{CenSet1} \;\triangleq\; \exists\, \mathit{aw} : \mathit{ICenSet1} \\[4pt]
\mathit{ICenSet1} \;\triangleq\; \mathit{Init} \wedge \Box[\mathit{NextSet1}]_v \\[4pt]
\quad v \;\triangleq\; \langle \mathit{inp},\ \mathit{disp},\ \mathit{aw},\ \mathit{old} \rangle \\[4pt]
\quad \mathit{InitSet} \;\triangleq\; \begin{array}[t]{l} \wedge\ \mathit{inp} = \mathit{NotArt} \\ \wedge\ \mathit{aw} = \{\,\} \\ \wedge\ \mathit{disp} \in \mathit{Art} \times \{0,1\} \\ \wedge\ \mathit{old} = \{\,\} \end{array} \\[4pt]
\quad \mathit{NextSet1} \;\triangleq\; \mathit{InputSet} \vee \mathit{DispOrNotSet} \vee \mathit{AckSet} \\[4pt]
\qquad \mathit{InputSet} \;\triangleq\; \begin{array}[t]{l} \wedge\ \mathit{inp} = \mathit{NotArt} \\ \wedge\ \mathit{inp}' \in \mathit{Art} \setminus \mathit{old} \\ \wedge\ \mathit{aw}' = \mathit{aw} \cup \{\mathit{inp}'\} \\ \wedge\ (\mathit{disp}' = \mathit{disp}) \wedge (\mathit{old}' = \mathit{old} \cup \{\mathit{inp}'\}) \end{array} \\[4pt]
\qquad \mathit{DispOrNotSet} \;\triangleq\; \exists\, w \in \mathit{aw} : \\
\qquad\qquad\qquad \begin{array}[t]{l} \wedge\ \begin{array}[t]{l} \vee\ \mathit{disp}' = \langle w,\ 1 - \mathit{disp}(2) \rangle \\ \vee\ \mathit{disp}' = \mathit{disp} \end{array} \\ \wedge\ \mathit{aw}' = \mathit{aw} \setminus \{w\} \\ \wedge\ (\mathit{inp}' = \mathit{inp}) \wedge (\mathit{old}' = \mathit{old}) \end{array} \\[4pt]
\qquad \mathit{AckSet} \;\triangleq\; \begin{array}[t]{l} \wedge\ \mathit{inp} \in \mathit{Art} \\ \wedge\ \mathit{inp}' = \mathit{NotArt} \\ \wedge\ (\mathit{aw}' = \mathit{aw}) \wedge (\mathit{disp}' = \mathit{disp}) \wedge (\mathit{old}' = \mathit{old}) \end{array}
\end{array}
$$

Figure 6.8: The program $\mathit{CenSet1}$.

To define $\mathit{DispOrNotSet}^p$, we define $\mathit{DorNSet}_i(w)$ as follows so $\mathit{DispOrNotSet}$ equals $\exists\, w \in \mathit{aw}, i \in \Pi : \mathit{DorNSet}_i(w)$.

$$
\mathit{DorNSet}_i(w) \;\triangleq\; \begin{array}[t]{l} \wedge\ \begin{array}[t]{l} \vee\ (i = \mathit{Yes}) \wedge (\mathit{disp}' = \langle w, 1 - \mathit{disp}(2) \rangle) \\ \vee\ (i = \mathit{No}) \ \ \wedge (\mathit{disp}' = \mathit{disp}) \end{array} \\ \wedge\ \mathit{aw}' = \mathit{aw} \setminus \{w\} \\ \wedge\ (\mathit{inp}' = \mathit{inp}) \wedge (\mathit{old}' = \mathit{old}) \end{array}
$$

The $\mathit{DispOrNotSet}^p$ action will have to erase the prediction by removing from the domain of p the picture being displayed or rejected. So let's define $\mathit{FcnMinus}(f, w)$ to equal the restriction of f to its domain minus the element w:

$$
\mathit{FcnMinus}(f, w) \;\triangleq\; x \in (\textsc{domain}(f) \setminus \{w\}) \mapsto f(x)
$$

We can now define:

$$\begin{aligned} &\mathit{DispOrNotSet}^p \;\triangleq \\ &\qquad \exists\, w \in aw : \mathit{DorNSet}_{p(w)}(w) \land (p' = \mathit{FcnMinus}(p, w)) \end{aligned}$$

Since AckSet^p neither makes nor satisfies a prediction, its definition is simply:

$$\mathit{AckSet}^p \;\triangleq\; \mathit{AckSet} \land (p' = p)$$

The rest of the definition of $\mathit{ICenSet1}^p$ should be clear.

We can then show that $\mathit{CenSet1}$ implements $\mathit{CenSet2}$ by showing

$$\models \mathit{ICenSet1}^p \;\Rightarrow\; (\mathit{ICenSet2} \text{ WITH } aw \leftarrow \mathit{awBar})$$

where awBar equals $\{w \in aw : p(w) = \mathit{Yes}\}$, the set of elements in aw that p predicts will be displayed.

We won't bother writing the generalization of Theorem 6.5 for a prophecy variable p that makes a set of predictions.

6.4.3.3 Further Generalizations

We now extract from our examples a more general formulation of prophecy variables. To construct a prophecy variable, we start with certain sets of actions we'll call *action sets*. For $\mathit{CenSeq1}$ there is one action set consisting of two actions: $\mathit{DispOrNotSeq}_{\mathit{Yes}}$, a $\mathit{DispOrNotSeq}$ action that displays the picture, and $\mathit{DispOrNotSeq}_{\mathit{No}}$, a $\mathit{DispOrNotSeq}$ action that doesn't display it. For $\mathit{CenSet1}$, for every $w \in \mathit{Art}$, there is an action set consisting of two actions: $\mathit{DispOrNotSet}_{w,\mathit{Yes}}$ and $\mathit{DDispOrNotSet}_{w,\mathit{No}}$, which are $\mathit{DispOrNotSet}$ actions that either display or don't display the input w. Thus, there is a set of action sets, one action set defined for each w in Art.

In general, for a program T, we have a set of action sets, each of which can be written as $\{A_i : i \in \Pi\}$. (The set Π can be different for different action sets.) That set of action sets is a constant; it doesn't change during a behavior of T.

A prophecy is a prediction about one of the program's action sets. That is, it is an element of the set Π defining the action set $\{A_i : i \in \Pi\}$. It predicts the value of i for which the next A_i step occurs. In any state, the value of a prophecy variable describes a set of predictions, some of which are active. For $\mathit{CenSeq1}$, the prophecy variable p equals a sequence of predictions, all for the same action set, only the first element of the sequence being active. For $\mathit{CenSet1}$, the prophecy variable p equals a set of prophecies, one for each action set defined by an element of aw, all of its prophecies being active.

In general, the program T^p is defined by adding the variable p in such a way that every behavior satisfying T^p satisfies T. Moreover, it ensures that no active prediction of p is ever violated. When an action predicted by an active prediction of p occurs, we say that the prediction is *fulfilled*. The initial value of p can contain prophecies. Prophecies can be added to or removed from p and/or made active or inactive by any action of T^p, so long as the following conditions are satisfied:

- No two active prophecies can be predictions for the same action set.
- For any action of T^p that adds a prophecy to p, it must be possible for the action to add a prophecy for any element of the set Π for that action set.
- An action of T^p that fulfills a prediction must remove that prediction from p. (It may also add one or more new predictions.)

The prophecy variables of $CenSeq1$ and $CenSet1$ made only predictions that were likely to be fulfilled. We could instead have used prophecy variables that a mathematician might consider simpler that make a lot more predictions. For $CenSeq1$, instead of having each $InputSeq$ step add a prediction to p, we could have let the initial value of p be an infinite sequence of predictions. The first element of the sequence would be the active one, and each $DispOrNotSeq$ action would remove that element from p. For $CenSet1$, we could have let the initial value of p be any element of $Art \rightarrow \{Yes, No\}$, predicting for each picture w whether or not it will be displayed if it is input. Since the same picture can't be input twice, the value of $p(inp)$ could be set by a $DispOrNotSet$ action to a value indicating that its prediction is inactive.

We could have used an even more extravagant prophecy variable for $CenSet1$—one that predicts not only whether each picture will be displayed or rejected, but in which order they will be input. The initial value of p would be an infinite sequence of predictions $\langle w, d \rangle$, for $w \in Art$ and $d \in \{Yes, No\}$, predicting not just if the next $DispOrNotSet$ step will display or reject the input, but that it must occur with inp equal to w. Almost all of those predictions will be impossible to fulfill because inp will not equal w. But as we've seen, impossible predictions don't matter because they just require the behavior to halt, which is either allowed or is ruled out by a liveness hypothesis. This may seem silly, but a prophecy variable that makes predictions that are almost all impossible is used in the example of Section 6.5 because it seems to provide the simplest way to define the needed refinement mapping.

6.4.4 The Existence of Refinement Mappings

We now state a completeness result for the auxiliary variables that we've described. Completeness means that if $\models T \Rightarrow \boldsymbol{\exists}\, \mathbf{y} : \mathit{IS}$ is true, where $\mathbf{y}$ is a list of variables, then we can successively add a list $\mathbf{a}$ of these auxiliary variables to T to obtain a formula $T^{\mathbf{a}}$ that implies IS under a refinement mapping that substitutes state expressions of $T^{\mathbf{a}}$ for the variables $\mathbf{y}$.

Like most such completeness results, it assumes that there is a mathematical proof of the result based on the semantics of the formulas. That is, we assume not only the truth of $\models T \Rightarrow \boldsymbol{\exists}\, \mathbf{y} : \mathit{IS}$, but that there exists a mathematical proof of its truth. Such a proof consists of an operator Φ such that for every behavior σ satisfying T, there is a behavior $\Phi(\sigma)$ satisfying IS that shows σ satisfies $\boldsymbol{\exists}\, \mathbf{y} : \mathit{IS}$. This means approximately that we can obtain $\Phi(\sigma)$ from σ by adding and/or removing stuttering steps and changing the values of the variables of $\mathbf{y}$. The precise statement of this condition is $\Phi(\sigma) \sim_{\mathbf{y}} \sigma$, where the definition of $\sim_{\mathbf{y}}$ is the same as that of $\sim_y$ in Section 6.1.3 with $=_y$ replaced by $=_{\mathbf{y}}$, and $s =_{\mathbf{y}} t$ defined to mean that states s and t are equal except perhaps for the values they assign to the variables of $\mathbf{y}$. The assumption that there exists a mathematical proof of $\models T \Rightarrow \boldsymbol{\exists}\, \mathbf{y} : \mathit{IS}$ is embodied in the use of Φ to construct the refinement mapping. Here is the precise statement of the theorem. It uses the notation that if $\mathbf{y}$ is the list $y_1, \ldots, y_n$ of variables and $\mathbf{exp}$ is the list $\mathit{exp}_1, \ldots, \mathit{exp}_n$ of expressions, then $\mathbf{y} \leftarrow \mathbf{exp}$ is an abbreviation for $y_1 \leftarrow \mathit{exp}_1, \ldots, y_n \leftarrow \mathit{exp}_n$.

Theorem 6.6 Let $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ be lists of variables, all distinct from one another; let the variables of T be $\mathbf{x}$ and $\mathbf{z}$ and the variables of IS be $\mathbf{x}$ and $\mathbf{y}$; and let T equal $\mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathbf{x}, \mathbf{z} \rangle} \land L$. Let the operator Φ map behaviors satisfying T to behaviors satisfying IS such that $\Phi(\sigma) \sim_y \sigma$. By adding history, stuttering, and prophecy variables to T, we can define a formula $T^{\mathbf{a}}$ such that $\boldsymbol{\exists}\, \mathbf{a} : T^{\mathbf{a}}$ is equivalent to T and a list $\mathbf{exp}$ of expressions defined in terms of Φ and the variables of $T^{\mathbf{a}}$ such that

$$\models T^{\mathbf{a}} \Rightarrow (\mathit{IS} \text{ WITH } \mathbf{y} \leftarrow \mathbf{exp})$$

The theorem makes no assumption about L other than that it contains no variables besides those of $\mathbf{x}$ and $\mathbf{z}$. It doesn't even have to be a liveness property.

Here is the idea behind the theorem's proof. We first add an infinite-stuttering variable t to avoid having to worry about terminating behaviors. We then add a history variable h that remembers the entire sequence of values of all the tuples $\langle \mathbf{x}, \mathbf{z}, t \rangle$ in all the states reached thus far, including the current one. We then add a prophecy sequence variable p that predicts the infinite sequence of all future values of $\langle \mathbf{x}, \mathbf{z}, t \rangle$. This means that in all states of the behavior, the value of $h \circ p$ is the entire sequence of values of $\langle \mathbf{x}, \mathbf{z}, t \rangle$ in the complete (infinite) behavior. Moreover, the length of h indicates the position of the current state in that behavior. The values of h and p and the mapping Φ provide all the information needed to determine the values to substitute for $\mathbf{y}$ to obtain a refinement mapping under which *IS* is simulated. The proof in the Appendix sketches the details.

The theorem shows that these auxiliary variables are, in principle, all we need to define a refinement mapping. It and its proof do not tell us how refinement mappings are defined in practice.

6.5 The FIFO Queue

This section presents a more realistic example of the use of auxiliary variables to show that one abstract program implements another. That makes it rather long, but it's included for two reasons. The first is that it describes serializability, which is an important concept for designing concurrent programs. The second is that stuttering and prophecy variables are not as intuitive as history variables, and a more realistic example may provide some insight into how they can be used in practice.

6.5.1 *Fifo* – A Linearizable Specification

Popular coding languages provide a small number of built-in data types such as finite-precision integers. Other data types must be implemented as objects. An object has a state and methods with which the program can read and modify parts of the state. We will ignore how objects are created and destroyed.

A simple example of such an object is a first in, first out queue, called a FIFO. We can think of the state of a FIFO as an ordinal sequence *queue* of elements from some set *Data*. A FIFO provides two methods, usually described as follows.

enqueue Takes an element of *Data* as an argument, and appends it to the end of *queue*. It returns no value.[1]

[1]Sometimes the queue can hold only some maximum number of elements, but for simplicity we assume that there is no such limit.

dequeue If *queue* is nonempty, it removes the first element of *queue* and returns it as the result. If *queue* is the empty sequence, it returns some special value.

An object is accessed only by executing its methods. For the purpose of correctness, the programmer needs to know nothing about how these two methods are implemented.

This kind of description is adequate for a method in a traditional program. It is inadequate for concurrent programs because it says nothing about what happens if two processes concurrently access the object. The call of a method and the return are usually described as single steps, but execution of the operation may consist of steps that occur between those two steps.

Often, it is considered an error if two processes concurrently access the same object. The object must either be accessed by only one process, or else accesses by different processes must be inside the critical section of a mutual exclusion algorithm. We're interested in objects that are meant to be accessed concurrently by multiple processes—for example, a critical section object for implementing mutual exclusion with *enter* and *exit* methods.

Maurice Herlihy and Jeannette Wing defined an object to be *linearizable* iff if acts as if the execution of a method consists of three steps: the call, the return, and between them a single step that performs the actual reading and/or modifying of the object's state [18]. The state of a linearizable object is described by internal variables. Only the call and return steps change interface variables. Linearizability has become a standard requirement for shared objects in concurrent systems.

We describe a linearizable FIFO as an abstract program. We require that execution of a *dequeue* operation when *queue* is empty waits for an element to be enqueued rather than returning a special value. This makes the FIFO more interesting because it involves process synchronization—making one process wait for another process to do something. Since the purpose of the example is to illustrate the use of auxiliary variables, which are added only to the safety property of a program, we consider only the safety property of a FIFO.

We assume there is a set *EnQers* of processes that perform *enqueue* operations. Execution of an *enqueue* operation by process e consists of three steps: a $BeginEnq(e)$ step that describes the call of the method, a $DoEnq(e)$ step that modifies the variable *queue*, and an $EndEnq(e)$ step that describes the return. The enqueuers communicate with the object through the interface variable *enq*, whose value is a function with domain *EnQers*.

The value of $enq(e)$ equals *Done* when enqueuer e is not performing an *enqueue* operation, and it equals the data value it is appending to *queue* when e is performing the operation, where *Done* is some constant not in *Data*. There is also an internal variable *enqInner*, where $enqInner(e)$ is set to *Busy* by the $BeginEnq(e)$ action and is set to *Done* by the $DoEnq(e)$ action.

Similarly, there is a set *DeQers* of dequeuer processes, each $d \in DeQers$ performing $BeginDeq(d)$, $DoDeq(d)$, and $EndDeq(d)$ steps. Dequeuers communicate with the object through the interface variable *deq*, where $deq(d)$ is set to *Busy* by the $BeginDeq(d)$ action and to the value that was dequeued by the $EndDeq(d)$ action. There is an internal variable *deqInner*, where $deqInner(d)$ is set to *Busy* by the $BeginDeq(d)$ action and set by $DoDeq(d)$ to the value dequeued by the *dequeue* operation. The complete definition of the abstract program is formula *Fifo* in Figure 6.9. It uses the UNCHANGED operator, where UNCHANGED *exp* equals $exp' = exp$. Thus, if v is a tuple $\langle v_1, \ldots, v_n \rangle$ of variables, UNCHANGED v asserts that $v'_i = v_i$ for all i in $1 \mathrel{..} n$.

6.5.2 *POFifo* – A More General Specification

6.5.2.1 The Background

Many people, myself included, used to believe that any implementation of a FIFO had to be a more concrete version of program *IFifo*, with the value of the variable *queue* encoded in the program's state. This implied that any implementation of a FIFO should implement *IFifo* under a refinement mapping, without having to add auxiliary variables.

We were wrong. Suppose two processes concurrently execute *enqueue* operations. When the two operations' *EndEnq* steps have occurred, program *IFifo* has appended both values to *queue* in some order, determining the order in which they will be dequeued. However, in a behavior of program *Fifo*, where *queue* is hidden, there is no way to know in which order the two values appear in *queue* until that order is revealed by *dequeue* operations. In their paper defining linearizability, Herlihy and Wing gave an algorithm that implements a FIFO in which, from a state immediately after both *enqueue* operations have completed, it is possible for the two values to be dequeued in either order by two successive non-concurrent *dequeue* operations. There is no queue encoded in the algorithm. While their algorithm implements *Fifo*, there is no refinement mapping under which it implements *IFifo* without the addition of auxiliary variables. In particular, showing that their algorithm

$$Fifo \triangleq \exists\!\!\exists\, queue, enqInner, deqInner : IFifo$$

$$IFifo \triangleq Init \land \Box[Next]_v$$

$$v \triangleq \langle enq, deq, queue, enqInner, deqInner \rangle$$

$$\begin{array}{lll}
Init & \triangleq & \land\ enq = (e \in EnQers \mapsto Done) \\
& & \land\ deq \in (DeQers \to Data) \\
& & \land\ queue = \langle\,\rangle \\
& & \land\ enqInner = (e \in EnQers \mapsto Done) \\
& & \land\ deqInner = deq
\end{array}$$

$$\begin{array}{lll}
Next & \triangleq & \lor\ \exists\, e \in EnQers : BeginEnq(e) \lor DoEnq(e) \lor EndEnq(e) \\
& & \lor\ \exists\, d \in DeQers : BeginDeq(d) \lor DoDeq(d) \lor EndDeq(d)
\end{array}$$

$$\begin{array}{lll}
BeginEnq(e) & \triangleq & \land\ enq(e) = Done \\
& & \land\ \exists\, D \in Data : enq' = (enq \text{ EXCEPT } e \mapsto D) \\
& & \land\ enqInner' = (enqInner \text{ EXCEPT } e \mapsto Busy) \\
& & \land\ \text{UNCHANGED } \langle deq, queue, deqInner \rangle
\end{array}$$

$$\begin{array}{lll}
DoEnq(e) & \triangleq & \land\ enqInner(e) = Busy \\
& & \land\ queue' = Append(queue, enq(e)) \\
& & \land\ enqInner' = (enqInner \text{ EXCEPT } e \mapsto Done) \\
& & \land\ \text{UNCHANGED } \langle deq, enq, deqInner \rangle
\end{array}$$

$$\begin{array}{lll}
EndEnq(e) & \triangleq & \land\ enq(e) \neq Done \\
& & \land\ enqInner(e) = Done \\
& & \land\ enq' = (enq \text{ EXCEPT } e \mapsto Done) \\
& & \land\ \text{UNCHANGED } \langle deq, queue, enqInner, deqInner \rangle
\end{array}$$

$$\begin{array}{lll}
BeginDeq(d) & \triangleq & \land\ deq(d) \neq Busy \\
& & \land\ deq' = (deq \text{ EXCEPT } d \mapsto Busy) \\
& & \land\ deqInner' = (deqInner \text{ EXCEPT } d \mapsto NoData) \\
& & \land\ \text{UNCHANGED } \langle enq, queue, enqInner \rangle
\end{array}$$

$$\begin{array}{lll}
DoDeq(d) & \triangleq & \land\ deq(d) = Busy \\
& & \land\ deqInner(d) = NoData \\
& & \land\ queue \neq \langle\,\rangle \\
& & \land\ deqInner' = (deqInner \text{ EXCEPT } d \mapsto Head(queue)) \\
& & \land\ queue' = Tail(queue) \\
& & \land\ \text{UNCHANGED } \langle enq, deq, enqInner \rangle
\end{array}$$

$$\begin{array}{lll}
EndDeq(d) & \triangleq & \land\ deq(d) = Busy \\
& & \land\ deqInner(d) \neq NoData \\
& & \land\ deq' = (deq \text{ EXCEPT } d \mapsto deqInner(d)) \\
& & \land\ \text{UNCHANGED } \langle enq, queue, enqInner, deqInner \rangle
\end{array}$$

Figure 6.9: The program *Fifo*.

implements *Fifo* requires adding a prophecy variable that predicts the order in which data items enqueued by concurrent *enqueue* operations will be dequeued.

What is encoded in the state of their algorithm is not a linearly ordered queue of enqueued data values, but rather a partial order on the set of enqueued values that indicates the possible orders in which the values can be returned by *dequeue* operations. A partial order on a set S is a relation $\prec$ on S that is transitive and has no cycles (which implies $a \not\prec a$ for any $a \in S$). For the partial ordering $\prec$ on the set of enqueued values, the relation $u \prec w$ means that value u must be dequeued before value w. Program *IFifo* is the special case in which that partial order is a total order, meaning that either $u \prec w$ or $w \prec u$ for any two distinct enqueued values u and w.

Presented here is a program *POFifo* that is equivalent to *Fifo*, but which is obtained by hiding internal variables in a program *IPOFifo* that maintains a partially ordered set of enqueued values rather than a queue. The Herlihy-Wing algorithm can be shown to implement *IPOFifo* under a refinement mapping defined in terms of its variables, without adding a prophecy variable.

6.5.2.2 Program *POFifo*

Execution of an operation is described by a sequence of steps. In a linearizable description, there are three steps: a step of the *Begin* action, a step of the *Do* action, and a step of the *End* action. One operation execution is defined to *precede* another if its *End* step precedes the other execution's *Begin* step. If neither of two operation executions precedes the other, then the executions are said to be *concurrent.*

Since *Begin* and *End* steps change interface variables, they appear in a behavior of *Fifo*. However, a *Do* step changes only internal variables, so it is not visible in a behavior of *Fifo*. If the executions of two *enqueues* are concurrent, then a behavior of *Fifo* does not show in which order their *Do* steps occurred in *IFifo*. Only if one *enqueue* execution precedes the other do we know from a behavior of *Fifo* the order in which the enqueued values appear in *queue*. Therefore, a behavior satisfies *Fifo* iff it satisfies the following two safety properties. They are stated informally, where a *value* is taken to mean a particular enqueueing of a data value.

F1. Each dequeued value has been enqueued, and an enqueued value is dequeued at most once.

F2. If an *enqueue* of a value v precedes an *enqueue* of a value w, then the

> *dequeue* of value w cannot precede the *dequeue* of value v.

The values enqueued by two concurrent *enqueue* executions may be dequeued in either order.

Program *IPOFifo* must maintain a set of enqueued items and some ordering relation among them. The first thing to observe is that the same data item might be enqueued twice before any item is dequeued. Since this is not a silly example, that possibility should be handled. An easy way to do that is to maintain a set of pairs $\langle d, i \rangle$ where d is the enqueued data value and i is an element of a set *Ids* of identifiers that serve to distinguish between different "copies" of an enqueued value. Let's call such a $\langle$data value, identifier pair$\rangle$ a *datum*. (We will use *datums* as the plural of *datum* because *data* suggests elements of *Data* rather than of $Data \times Id$. We will continue to call an element of *Data* a *data value*.)

IPOFifo will use an internal variable *elts* whose value is the set of currently enqueued datums. It will also have an internal variable whose value is an ordering relation $\prec$ on the set *elts* where, $u \prec w$ means that datum u must be dequeued before datum w is. For *IPOFifo* to describe a linearizable object, execution of an *enqueue* operation must consist of a *BeginPOEnq* step, followed by a *DoPOEnq* step that puts the datum in *elts*, followed by an *EndPOEnq* step. Condition F2 is satisfied if the *DoPOEnq* step that puts a datum w in *elts* adds a relation $u \prec w$ for every datum u put in *elts* by an *enqueue* operation that has completed. Of course, F1 is satisfied if every *DoPODeq* step obtains its data item from a datum that it removes from *elts*.

Given any behavior σ satisfying *IPOFifo*, we can obtain a behavior τ satisfying *IPOFifo* by moving every *DoPOEnq* step earlier in the behavior so it occurs immediately after its operation's *BeginPOEnq* step, and moving every *DoPODeq* step later in the behavior so it occurs immediately before its operation's *EndPODeq* step. Moreover, since the *Do*... steps change only internal variables, the values of the interface variables are the same in each state of τ as in σ. Therefore, without eliminating any possible behaviors of *POFifo* (which are obtained by hiding the internal variables of *IPOFifo*), we can require that a *BeginPOEnq* step be immediately followed by the operation's *DoPOEnq* step, and that an *EndPODeq* step be immediately preceded by the operation's *DoPODeq* step. This means that there's no need for the *Do*... actions; the *BeginPOEnq* and *DoPOEnq* actions can be combined into a single action, as can the *DoPODeq* and *EndPODeq* actions. We will therefore simplify *IPOFifo* be eliminating the *Do*... actions and having only *Begin*... and *End*... actions.

We can now write the program *IPOFifo*. It will have the same constants as *IFifo* plus the set *Ids* of identifiers; and it will have the same interface variables *deq*. It will have the internal variable *elts* whose value is the set of currently enqueued datums.

IPOFifo will need an internal variable to describe the partial order $\prec$ on the set *elts*. Mathematicians describe a relation $\prec$ on a set S as a subset of $S \times S$, where $u \prec v$ is an abbreviation for $\langle u, v \rangle \in \prec$. We'll do the same thing, except it would be confusing to use the symbol $\prec$ as a variable. We will therefore let *before* be the variable whose value is a subset of $elts \times elts$ such that $u \prec v$ means $\langle u, v \rangle \in before$.

Finally, when enqueueing a datum w, the *BeginPOEnq* step must add to $\prec$ the relation $u \prec w$ for a datum u in *elts* iff the *enqueue* operation that added u has completed. That information is not contained in the interface variable *enq* because $enq(e)$ contains only the data value that an uncompleted *enqueue* operation is enqueueing, not which datum the operation put in *elts*. Therefore, we add to *IPOFifo* an internal variable *adding* such that $adding(e)$ equals the datum in *elts* that enqueuer e put in *elts*, and equals a value *NonElt* that is not a datum if e is not currently performing an *enqueue* operation.

We use *adding* to define the following state expression, whose value is the set of datums enqueued by operations whose executions have not yet completed:

$$beingAdded \;\triangleq\; \{adding(e) : e \in EnQers\} \setminus \{NonElt\}$$

The set *beingAdded* need not be a subset of *elts* because it can contain datums that were removed from *elts* by *dequeue* operations before the operations that enqueued them have completed.

The program *POFifo* is defined in Figure 6.10. Here are explanations of the four disjuncts of the next-state action *PONext*.

$BeginPOEnq(e)$ Enabled when $enq(e) = Done$, it:

- Sets $enq(e)$ to the data value D that e is enqueuing.
- Adds a datum $\langle D, id \rangle$ to *elts* for some $id \in Ids$ for which $\langle D, id \rangle$ is not already in *elts* or *beingAdded*.
- Modifies *before* to add the relations $el \prec \langle D, id \rangle$ for every el in *elts* that is not in *beingAdded*.
- Sets $adding(e)$ to $\langle D, id \rangle$, thereby adding $\langle D, id \rangle$ to *beingAdded*.

$$POFifo \triangleq \boldsymbol{\exists}\, elts, before, adding : IPOFifo$$

$$IPOFifo \triangleq POInit \land \Box[PONext]_{POv}$$

$$POv \triangleq \langle enq, deq, elts, before, adding \rangle$$

$$\begin{aligned} POInit \triangleq\ & \land\ enq = (e \in EnQers \mapsto Done) \\ & \land\ deq \in (DeQers \to Data) \\ & \land\ elts = \{\} \\ & \land\ before = \{\} \\ & \land\ adding = (e \in EnQers \mapsto NonElt) \end{aligned}$$

$$\begin{aligned} PONext \triangleq\ & \lor\ \exists\, e \in EnQers : BeginPOEnq(e) \lor EndPOEnq(e) \\ & \lor\ \exists\, d \in DeQers : BeginPODeq(d) \lor EndPODeq(d) \end{aligned}$$

$$\begin{aligned} & BeginPOEnq(e) \triangleq \\ & \qquad \land\ enq(e) = Done \\ & \qquad \land\ \exists\, D \in Data : \exists\, id \in \{i \in Ids : \langle D, i \rangle \notin (elts \cup beingAdded)\} : \\ & \qquad\qquad \land\ enq' = (enq \text{ EXCEPT } e \mapsto D) \\ & \qquad\qquad \land\ elts' = elts \cup \{\langle D, id \rangle\} \\ & \qquad\qquad \land\ before' = before \cup \{\langle el, \langle D, id \rangle\rangle : el \in (elts \setminus beingAdded)\} \\ & \qquad\qquad \land\ adding' = (adding \text{ EXCEPT } e \mapsto \langle D, id \rangle) \\ & \qquad \land\ deq' = deq \end{aligned}$$

$$\begin{aligned} EndPOEnq(e) \triangleq\ & \land\ enq(e) \neq Done \\ & \land\ enq' = (enq \text{ EXCEPT } e \mapsto Done) \\ & \land\ adding' = (adding \text{ EXCEPT } e \mapsto NonElt) \\ & \land\ \text{UNCHANGED } \langle deq, elts, before \rangle \end{aligned}$$

$$\begin{aligned} BeginPODeq(d) \triangleq\ & \land\ deq(d) \neq Busy \\ & \land\ deq' = (deq \text{ EXCEPT } d \mapsto Busy) \\ & \land\ \text{UNCHANGED } \langle enq, elts, before, adding \rangle \end{aligned}$$

$$\begin{aligned} EndPODeq(d) \triangleq\ & \land\ deq(d) = Busy \\ & \land\ \exists\, el \in elts : \\ & \qquad \land\ \forall\, el2 \in elts : \neg(el2 \prec el) \\ & \qquad \land\ elts' = elts \setminus \{el\} \\ & \qquad \land\ deq' = (deq \text{ EXCEPT } d \mapsto el(1)) \\ & \qquad \land\ before' = before \cap (elts' \times elts') \\ & \land\ \text{UNCHANGED } \langle enq, adding \rangle \end{aligned}$$

Figure 6.10: The program *POFifo*.

EndPOEnq(e) Enabled when *enq*(e) is a data value, it sets *enq*(e) to *Done* and sets *adding*(e) to *NonElt*, thereby removing from *beingAdded* the datum that e had enqueued.

BeginPODeq(d) Enabled when *deq*(d) is a data value, it sets *deq*(d) to *Busy*.

EndPODeq(d) Enabled when *deq*(d) equals *Busy* and *elts* is not empty, which implies that *elts* contains at least one minimal datum (a datum not preceded in the $\prec$ relation by any other datum in *elts*), since the datum most recently added to *elts* must be a minimal datum. The action chooses an arbitrary minimal datum *el* of *elts*, removes it from *elts*, sets *deq*(d) to its data value component, and modifies *before* to remove all relations $el \prec el2$ for elements $el2$ of *elts*.

6.5.3 Showing *IPOFifo* Implements *Fifo*

Formulas *POFifo* and *Fifo* are equivalent. However, more important and more interesting than showing that *Fifo* refines *POFifo* is showing that *POFifo* refines *Fifo*. It's more important because *Fifo* is the generally accepted description of a FIFO. By showing that *POFifo* implements *Fifo*, we can show that an algorithm that doesn't maintain a totally ordered queue implements *Fifo* by showing that it implements *POFifo*. It's more interesting because showing that *POFifo* implements *Fifo* requires all three kinds of auxiliary variables. So, the problem of showing that *Fifo* refines *POFifo* is not discussed.

To show that *POFifo* refines *Fifo*, we have to define a refinement mapping under which *IPOFifo* implements *IFifo*. This requires adding to *IPOFifo* first a prophecy variable, then a history variable, then a stuttering variable. They are added in the three following subsections. Adding the prophecy variable is straightforward and the necessary details are presented. How to add the other two variables and define the refinement mapping are then sketched. Rigorously defining those two programs and the refinement mapping would be a marvelous learning experience. However, the chance of doing it without making any error is small unless you use a tool to check what you have done.

6.5.3.1 The Prophecy Variable

The most significant problem in showing that *POFifo* implements *Fifo* is that *IFifo* decides the order in which concurrently enqueued values will be dequeued before *IPOFifo* does. This tells us that to define a refinement mapping under which *IFifo* is implemented by *POFifo*, we need to add a prophecy variable p to *IPOFifo*.

The simplest way I know of making the necessary predictions is with a prophecy sequence variable p that predicts the sequence of datums that will be dequeued next. The first item in the sequence predicts the datum that the next *EndPODeq* step removes from *elts*, and that step of course removes the prediction from p. The natural step to append a prediction to the sequence p is the *BeginPOEnq* step that adds a datum to *elts*. The length of p therefore always equals the number of datums in *elts*. The predictions are completely arbitrary datums, so in almost all behaviors they will be impossible to fulfill, at some point not allowing any more datums to be dequeued. But as we've seen, unfulfillable predictions are no problem. Since no predictions are made about *enqueue* operations, they can keep being performed even if the datums they enqueue can never be dequeued.

The set Π of possible predictions equals the set $Data \times Ids$ of all possible datums. The $BeginPOEnq(e)$ action should append a prediction to p, and the $EndPOEnq(e)$ and $BeginPODeq(d)$ actions should leave p unchanged. We can therefore define three of the four subactions of the next-state action $PONext^p$ of $IPOFifo^p$ by:

$$\begin{aligned} BeginPOEnq^p(e) \;\triangleq\;& \land\; BeginPOEnq(e) \\ & \land\; \exists\, el \in Data \times Ids \,:\, p' = Append(p, el) \end{aligned}$$

$$EndPOEnq^p(e) \;\triangleq\; EndPOEnq(e) \land (p' = p)$$

$$BeginPODeq^p(d) \;\triangleq\; BeginPODeq(d) \land (p' = p)$$

The prediction made by the first item $p(1)$ of the sequence p is the datum that the next $EndPODeq(d)$ step will remove from *elts*. The datum $p(1)$ is removed by this step iff $elts' = elts \setminus \{p(1)\}$ is true of the step. Since the step must remove the prediction, we can define:

$$\begin{aligned} & EndPODeq^p(d) \;\triangleq \\ & \quad EndPODeq(d) \land (elts' = elts \setminus \{p(1)\}) \land (p' = Tail(p)) \end{aligned}$$

These four action definitions, and the observation that $Init^p$ should equal $Init \land (p = \langle\,\rangle)$ give us all the pieces of the definition of $IPOFifo^p$. I won't bother to put them together.

6.5.3.2 The History Variable *qBar*

We now must decide how to define *queueBar*, the state expression that is substituted for *queue* in the refinement mapping. It's easiest to think of defining *queueBar* as the sequence not of the data values to be substituted for the variable *queue* of *IFifo*, but of the sequence of datums being dequeued from *elts* by *POFifo*. We can then define the refinement mapping so it substitutes for the variable *queue* of *IFifo* the sequence of data values in the datums of *queueBar*.

Since there are no *Do* . . . steps in *IPOFifo*, we will have to add stuttering steps to append datums to *queueBar*. To do that, we will first add to $IPOFifo^{p}$ a history variable *qBar* to produce a program we'll call $IPOFifo^{pq}$. The value of *qBar* will be the sequence of datums that will appear in *queueBar*. However, datums will be appended to *qBar* by *Begin* . . . and *End* . . . steps, so stuttering steps will then have to be added to $IPOFifo^{pq}$ that append the datums in *qBar* one at a time to *queueBar*, since *IFifo* appends data values to *queue* one at a time. How those stuttering steps are added is explained in Section 6.5.3.3.

We will see that a single $BeginPOEnq^{pq}$ step may append multiple datums to *qBar*, so letting *IPOFifo* have *Do* . . . steps wouldn't have eliminated the need to add stuttering steps. The following definition of the history variable *qBar* is subtle. To help you understand it, I suggest you check how the definition works on the first few steps of several different behaviors of $IPOFifo^{p}$.

Suppose that program $BeginPOEnq^{p}$ is in a state in which it's possible for a *dequeue* operation to remove a datum from *elts*. For *qBar* to describe the datums that are in *queueBar*, the datum removed by that *dequeue* operation must be in *qBar*. An $EndPODeq^{p}$ step can remove only the datum $p(1)$ from *elts*, and it can remove $p(1)$ iff $p(1)$ is a minimal element for the relation $\prec$ of *elts*. Therefore, $p(1)$ must be the first element of *qBar* iff $p(1)$ is a minimal element of *elts*. If $p(1)$ can be *dequeued*, then $p(2)$ can be dequeued after it iff $p(2)$ is a minimal element of $elts \setminus \{p(1)\}$. And so on.

Define the state predicate *pg* to be the longest prefix of *p* all of whose datums can be dequeued from *elts* if no further datums are enqueued in *elts*. It follows from the definition of $IPOFifo^{p}$ that *pg* equals the longest prefix of *p* satisfying the following conditions:

Q1. Every datum in *pg* is in *elts*.

Q2. No datum appears twice in *pg*.

Q3. For each $i \in 1 \,..\, Len(pg)$ and each datum $u \in elts$, if $u \prec pg(i)$ then $u = pg(j)$ for some $j \in 1 \,..\, (i-1)$.

We have shown that *pg* must be a prefix of *qBar*. Our strategy for defining $IPOFifo^{pq}$ by adding the history variable *qBar* to $IPOFifo^{p}$ is to keep *qBar* equal to *pg* for as long as possible. To see how to do that, let's see how *pg* can change.

The sequence *pg* can become shorter only when an $EndPODeq^p$ step occurs, in which case *p* is not the empty sequence and *pg* is a nonempty prefix of *p*. The step removes the first element of *p* and *pg*, so $p' = Tail(p)$, $pg' = Tail(pg)$, and $qBar' = Tail(qBar)$.

The sequence *pg* can be made longer by a $BeginPOEnq^p$ step as follows. Suppose the step appends the prediction *u* to *p* and adds the datum *w* to *elts*. The value of *pg* at the beginning of the step is a proper prefix of $p \circ \langle u \rangle$. If *w* equals the prediction in $p \circ \langle u \rangle$ immediately after *pg*, then *w* will be appended to *pg* iff doing so would not violate Q3. (We'll see in a moment when it would violate Q3.) If *w* can be appended to *pg* and the prediction following *w* in *p* is already in *elts*, then it might be possible to append that datum to *pg* as well. And so on. Thus, it's possible for the $BeginPOEnq^p$ step to append several datums to *pg*. If our strategy has been successful thus far and $qBar = pq$ at the beginning of the step, then a $BeginPOEnq^{pq}$ step implies $qBar' = pq'$. This makes *qBar* a prefix of $qBar'$, as it should be because stuttering steps to be added after a $BeginPOEnq^p$ step should change *queueBar* only by appending datums to it.

There is one situation in which it is impossible for any further datum to be appended to *pg*. One or more datums can be appended to *pg* only by a $BeginPOEnq^p$ that adds a datum *w* to *elts* that can be appended to *pg*. However if there is a datum *u* in *elts* that is neither in the sequence *pg* nor in the set *beingAdded*, then adding *w* to *elts* also adds the relation $u \prec w$. This relation means that *w* can't be appended to *pg* because that would violate condition Q3. Thus, if there is a datum *u* in *elts* that is neither in *pg* nor *beingAdded*, then no datums can be added to *pg*. Moreover, the datum *u* can never be removed from *elts* because it is not in *pg* and can never be in *pg* because no more datums can be added to *pg*. (The datum *u* can't be added to *beingAdded* because a *BeginPOEnq* step can't add a datum to *elts* that is already in *elts*.) Let's call a state in which there is a datum in *elts* that is not in *beingAdded* a *blocked* state. In a blocked state, datums can be removed from the head of *pq* by $EndPODeq^p$ steps, but no new datums can be appended to *pq*. So, if and when enough $EndPODeq^p$ steps have occurred to remove all the datums from *pq*, then no more $EndPODeq^p$ steps can occur. That means that any further *enqueue* operations that are begun with a $BeginPODeq^p$ step must block, never able to complete.

Let's consider the first step that caused a blocked state—that is, causing there to be an element *u* in *elts* that is neither in *pg* nor *beingAdded*. Since *u* was added to *elts* by a $BeginPOEnq^p$ step that put *u* in *beingAdded*, it must be the $EndPOEnq^p$ step of the *enqueue* operation that added u to *elts* that caused the blocked state by removing u from *beingAdded*. Until that blocked state was reached, *qBar* equaled *pg*. However, since u has not been dequeued, it must be in *queueBar* after that $EndPOEnq^p$ step because that step must implement the *EndEnq* step of *IFifo*. Thus that $EndPOEnq^p$ step must append u to *qBar*. Therefore, the first blocked state is the first state in which $qBar \neq pg$. In that state, *qBar* equals $pg \circ \langle u \rangle$.

From that first blocked state on, no new datums can be added to *pg*, so the datum u can never be dequeued. Therefore, whenever an $EndPOEnq^p$ step occurs for an operation that enqueued a datum w, if w is in *elts* (so it hasn't been dequeued) and is not in *pq*, then that $EndPOEnq^p$ step must append w to *qBar*.

To recapitulate, here is how we add the history variable *qBar* to $IPOFifo^p$ to obtain the program $IPOFifo^{pq}$. These rules imply that, at any point in the behavior, *qBar* will equal $pg \circ eb$ where *pg* is the state function of $IPOFifo^p$ defined above and *eb* is a sequence of datums in *elts* that are not in *pg*. Initially, *pg* and *eb* equal $\langle\, \rangle$.

- An $EndPODeq^p$ step can occur only if $pg \neq \langle\, \rangle$. Such a step satisfies $pg' = Tail(pg)$ and $qBar' = Tail(qBar)$.
- An $EndPOEnq^p$ step that removes from *beingAdded* a datum w that is in *elts* but not in *pg* appends w to *eb* and hence to *qBar*.
- A $BeginPOEnq^p$ step that occurs when $eb = \langle\, \rangle$ (so $qBar = pg$) sets $qBar'$ equal to pg'.

These rules imply that a datum can never be removed from *eb*, so once *eb* is nonempty no new datums can be added to *pg* and only datums currently in *pg* can ever be dequeued.

Observe that the sequence *pg* and the set of datums in *eb* can be defined in terms of the variables of $IPOFifo^p$. A history variable is needed only to remember the order in which datums have been appended to *eb*. This suggests that it's a little simpler to make *eb* the history variable and define *qbar* to be the state expression $pg \circ eb$. However, I could not have discovered this without first understanding how *qBar* should be defined.

Writing a complete definition of $IPOFifo^{pq}$ is straightforward, once we have solved the problem of writing a precise mathematical definition of the state function *qBar* in terms of the variables of $IPOFifo^p$. (It took me a few tries to get that definition right, using a model checker to find my mistakes.) That definition is omitted.

6.5.3.3 Stuttering and the Refinement Mapping

Having defined *qBar* for $IPOFifo^{pq}$, adding a stuttering variable s and defining the refinement mapping under which the resulting program $IPOFifo^{pqs}$ implements *IFifo* are comparatively straightforward. First, s has to add stuttering steps that allow us to define *queueBar*. Recall that *qBar* is defined so it is changed by interface actions (the only ones that $IPOFifo^{pq}$ has) the way internal *Do* ... actions of *IFifo* change the value of the internal variable *queue* of *IFifo*. Since *queueBar* should implement *queue* under the refinement mapping (except that *queueBar* contains datums not just the data values), it needs to be changed by stuttering steps added to $IPOFifo^{pq}$. We define the program $IPOFifo^{pqs}$ by adding a stuttering variable s that adds steps in the following three cases. We define *queueBar* to be the state expression that equals *qBar* except as noted below.

1. s adds a single stuttering step before each $EndPODeq^{pq}$ step. The value of *queueBar* equals $Tail(qBar)$ immediately after that stuttering step.

2. s adds a stuttering step before each $EndPOEnq^{pq}$ step that appends an element w to eb (and hence to *qBar*). Immediately after that stuttering step, *queueBar* equals $qBar \circ \langle w \rangle$.

3. Following each $BeginPOEnq^{pq}$ step such that $Len(pg') > Len(pg)$ (which implies $eq = \langle\,\rangle$), s adds $Len(pg') - Len(pg)$ stuttering steps. While there are k more of those stuttering steps left to be executed, *queueBar* equals the sequence obtained by removing the last k elements of *qBar*.

Encoding in the value of the stuttering variable s for which of the three cases the variable is being added, and in case 2 for which enqueuer e the step is an $EndPOEnq^{pq}(e)$ step, allows the value of *queueBar* to be defined in terms of the values of s, *qBar*, and (for case 2) *adding*.

We still have to define the state functions *enqInnerBar* and *deqInnerBar* that are substituted for *enqInner* and *deqInner* in the refinement mapping under which $IPOFifo^{pqs}$ implements *IFifo*. The value of $enqInnerBar(e)$ for an enqueuer e should equal *Done* except when $adding(e)$ equals the datum that e is enqueueing, and that datum is not yet in *queueBar*. This means that *enqInnerBar* can be defined in terms of *adding* and *queueBar*.

The value of $deqInnerBar(d)$ for a dequeuer d should equal the value of $deq(d)$ except between when d has removed the first element of *queueBar* (by executing the stuttering step added in case 1) and before the subsequent $EndPODeq^{pqs}(d)$ step has occurred. In that case, $deq(d)$ should equal $qBar(1)$. It's therefore easy to define *deqInnerBar* as a state function of $IPOFifo^{pqs}$ if the value of the stuttering variable s added in case 1 contains the value of d for which the following $EndPODeq^{pqs}(d)$ step is to be performed.

This completes the sketch of how auxiliary variables are added to *IPOFifo* to define a refinement mapping under which it implies *IFifo*, showing that *POFifo* refines *Fifo*. The intellectually challenging part was discovering how to define *qBar*. It took me quite a bit of thinking to find the definition. This was not surprising. The example of the FIFO had been studied for at least 15 years before Herlihy and Wing discovered that it could be implemented without maintaining a totally ordered queue. Given the definition of *qBar*, constructing the refinement mapping required the ability to write abstract programs mathematically—an ability that comes with practice.

6.6 Prophecy Constants

We have seen examples of showing $\models T \Rightarrow S$, where S equals $\exists \ldots : IS$, by adding an auxiliary variable a to T and showing T^a implies IS under a refinement mapping. In these examples, IS and S were safety properties. Liveness was shown by showing that T^a conjoined with the lower-level program's liveness property implies the higher-level program's liveness property.

Let's return to an example we considered in Section 6.1.3. Define IS and T to be the two formulas defined in (6.7) in (6.8):

$$\begin{aligned} IS \;&\triangleq\; \land\, (x = 0) \land (y \in \mathbb{N}) \\ &\;\; \land\, \Box\, [(y > 0) \land (x' = x + 1) \land (y' = y - 1)]_{\langle x, y \rangle} \\ T \;&\triangleq\; (x = 0) \,\land\, \Box[x' = x + 1]_x \,\land\, \Diamond\Box[x' = x]_x \end{aligned}$$

and let S equal $\exists\, y : IS$. We observed that formulas S and T are equivalent, so $\models T \Rightarrow S$ is true. To define a refinement mapping under which T implies IS, we have to add one or more auxiliary variables to T to obtain an expression to substitute for y.

The liveness property $\Diamond\Box[x' = x]_x$ implies that a behavior of T eventually stops incrementing x and terminates. To define the refinement mapping, we need a variable that can predict the value of x when the behavior terminates. So, we need a prophecy variable.

Because we have to predict something that is implied by a liveness property, it appears that the only kind of prophecy variable that will work is one that predicts the entire future—namely an infinite prophecy variable that predicts all the future values of x. This means adding auxiliary variables, including a prophecy variable, the way it is done in the completeness proof of Section 6.4.4. This is disturbingly complicated for such a simple example.

Fortunately, there is a simple way to construct the refinement mapping under which T implies IS without using a prophecy variable. We first observe that T implies that eventually x is forever equal to some natural number—more precisely:

$$(6.17) \quad \models T \;\Rightarrow\; \exists\, n \in \mathbb{N} \,:\, \Diamond\Box(x = n)$$

This implies:

$$\models T \;\equiv\; \exists\, n \in \mathbb{N} \,:\, T \land \Diamond\Box(x = n)$$

For any formulas F and G and constants c and C, to prove $(\exists\, c \in C : F) \Rightarrow G$ it suffices to prove $(c \in C) \land F \Rightarrow G$. Therefore, to prove $T \Rightarrow IS$, it suffices to prove

$$(6.18) \quad \models (n \in \mathbb{N}) \,\land\, \Diamond\Box(x = n) \,\land\, T \;\Rightarrow\; IS$$

We can do this by adding the history variable h as follows to obtain T^h:

$$\begin{aligned} T^h \;\triangleq\;\; & \land\; (n \in \mathbb{N}) \,\land\, (x = 0) \,\land\, (h = n) \\ & \land\; \Box[(x' = x + 1) \,\land\, (h' = h - 1)]_{\langle x, h \rangle} \\ & \land\; \Diamond\Box[x' = x]_x \,\land\, \Diamond\Box(x = n) \end{aligned}$$

We can then use the refinement mapping that substitutes h for y, and we show:

$$\models T^h \;\Rightarrow\; (IS \text{ WITH } y \leftarrow h)$$

The key step in what we've done is the use of (6.17) to reduce showing $\models T \Rightarrow IS$ to showing (6.18). This procedure is called adding the *prophecy constant* n to T. This is perhaps a silly name, since n is just an ordinary bound constant. However, it serves to predict something that is eventually going to happen—in this case, the final value x will have.

In general, we add a prophecy constant c to a program T by showing $\models T \Rightarrow \exists\, c \in C : L$, where the constant c does not occur in T, which implies

$$\models T \;\equiv\; \exists\, c \in C \,:\, T \land L$$

and showing $\models T \Rightarrow IS$ by showing

$$\models (c \in C) \wedge T \wedge L \;\Rightarrow\; IC$$

It can be shown that prophecy constants are in principle as powerful as prophecy variables. In particular, Theorem 6.6 of Section 6.4.4 is true with the prophecy variable replaced by a prophecy constant. This is proved by modifying the proof of Theorem 6.6 as described in Appendix Section B.12, after the proof of that theorem.

Although in theory equivalent, prophecy variables and prophecy constants are quite different in practice. It appears that a prophecy variable is best for predicting what a safety property implies may happen, while a prophecy constant is best for predicting what a liveness property implies must eventually happen.

Prophecy constants were introduced by Wim Hesselink, who called them "eternity variables" [19]. He did not represent programs mathematically, so he had to invent a rule for adding them to programs.

Chapter 7

Loose Ends

This chapter covers two topics that, to my knowledge, have not yet seen any industrial application. However, they might in the future become useful. The first topic is reduction, which is about verifying that a program satisfies a property by verifying that a coarser-grained version of the program satisfies it. Even if you never use it, understanding the principles behind reduction can help you choose the appropriate grain of atomicity for abstract programs. For that purpose, skimming through sections 7.1.1–7.1.3 should suffice.

The second topic is about representing a program as the composition of component programs. We have been representing the components that make up a program, such as the individual processes in a multiprocess program, as disjuncts of the next-state action. Section 7.2 explains how the components that form a program can be described as programs. How this is done depends on why it is done. Two reasons for doing it and the methods they lead to are presented.

7.1 Reduction

7.1.1 Introduction

7.1.1.1 What Reduction Is

When writing an abstract program to describe some aspect of a concrete one, we must decide what constitutes a single step of a behavior. Stated another way, we must describe what the grain of atomicity of the next-state action should be. The only advice provided thus far is that we should use the coarsest grain of atomicity (the fewest steps) that is a sufficiently accurate representation of that aspect of the concrete program. "Sufficiently accurate" means that either we believe it is easy to make the concrete program implement that grain of atomicity, or we are deferring the problem of how those atomic actions are implemented.

Some work has addressed the problem of formalizing what makes an abstract program "sufficiently accurate", starting with a 1975 paper by Richard Lipton [39]. This work used the approach called *reduction*, which replaces a program S with an "equivalent" coarser-grained program S^R called the *reduced version* of S. Certain properties of S are verified by showing that S^R satisfies them. The program S^R is obtained from S by replacing certain nonatomic operations with atomic actions, each atomic action producing the same effect as executing all the steps of the nonatomic operation it replaces one after another, with no intervening steps of other operations. The reduced program S^R is therefore simpler and easier to reason about than program S.

It was never clear in exactly what sense S^R was equivalent to S, and the results were restricted to particular classes of programs and of the properties that could be verified. TLA enabled a new way of viewing reduction. In that view, the variables of S are replaced in S^R by "virtual" variables, and S implements S^R under a refinement mapping. The refinement mapping is not made explicit, but the relation between the values of the actual and the virtual variables is described by an invariant. I believe that this mathematical view encompasses all previous work on reduction for concurrent programs.

Our basic problem approach to writing a correct concrete program is showing that it refines an abstract program. There are two aspects to refining one program with another: data refinement and step refinement. Modern coding languages have made data refinement easier by providing higher-level, more abstract data structures. It is now almost as easy to write a program that manipulates integers as one that manipulates bit strings representing a finite set of integers. There has been much less progress in making step refinement easier. As explained in Section 6.5.1, a linearizable object allows a coarse grain of atomicity in descriptions of the code that executes operations on the object. However, the only general method of implementing a linearizable object still seems to be the one invented by Dijkstra in the 1960s: putting the code that reads and/or modifies the state of the object in a critical section.

I believe that better ways of describing the grain of atomicity will be needed if rigorous verification that concrete concurrent programs implement abstract ones is to become common practice. Reduction may provide the key to doing this. This section provides a mathematical foundation for understanding reduction. The theorems presented here are not the most general ones possible; some generalizations can be found elsewhere [8]. Also omitted are rigorous proofs. I know of no industrial use of reduction or of tools to support it; and I have no experience using the results described here in practice. The problem it addresses is real, but I don't know if reduction is the solution.

7.1.1.2 The TLA Approach

Here is how reduction can be described in TLA. Let $\mathbf{x}$ be the variables x_1, $\ldots$, x_n of S. Reduction is usually described by letting $\mathbf{x}$ also be the variables of S^R. However, I find that reduction is easier to understand by letting the variables of S^R be different from those of S. We will let those variables be the list $\mathbf{X}$ of variables $X_1, \ldots, X_n$. The goal is then to verify that S satisfies a property P by verifying that S^R satisfies the property P WITH $\mathbf{x} \leftarrow \mathbf{X}$.

To deduce $\models S \Rightarrow P$, where S and P contain the variables $\mathbf{x}$, from $\models S^R \Rightarrow (P \text{ WITH } \mathbf{x} \leftarrow \mathbf{X})$, we need a relation between the variables $\mathbf{x}$ and the variables $\mathbf{X}$. That relation is expressed by a state predicate I^R containing the variables $\mathbf{x}$ and $\mathbf{X}$. We then deduce that S satisfies P from the following three conditions. The formula T will be explained later; ignore it for the moment.

R1. $\models S \wedge T \;\Rightarrow\; \exists\, \mathbf{X} : S^R \wedge \Box I^R$

R2. $\models S^R \;\Rightarrow\; (P \text{ WITH } \mathbf{x} \leftarrow \mathbf{X})$

R3. $\models (P \text{ WITH } \mathbf{x} \leftarrow \mathbf{X}) \wedge \Box I^R \;\Rightarrow\; P$

Condition R1 is implied by theorems whose hypotheses are the conditions necessary for reduction to be possible. Condition R2 asserts that P with its variables $\mathbf{x}$ replaced by $\mathbf{X}$ is satisfied by the reduced program S^R, which should be easier to verify than that P is satisfied by the finer-grained program S.

Throughout this section on reduction, we abbreviate F WITH $\mathbf{x} \leftarrow \mathbf{X}$ as $\overline{F}$, for any formula F. Thus R2 and R3 can be written as

$$\models S^R \;\Rightarrow\; \overline{P} \quad \text{and} \quad \models \overline{P} \wedge \Box I^R \;\Rightarrow\; P$$

We first consider the case in which S is the usual TLA safety property $\mathit{Init} \wedge \Box[\mathit{Next}]_{\langle \mathbf{x} \rangle}$ for an abstract program. We then consider the program described by $S \wedge F$, where F is the conjunction of fairness properties for S. Conditions R1–R3 will then have S replaced by $S \wedge F$, the reduced program $(S \wedge F)^R$ being defined to equal $S^R \wedge F^R$ where F^R is obtained by replacing each fairness condition for an action A of S with fairness of a corresponding action A^R of S^R.

The formula T in the hypothesis of R1 is a liveness assumption that will be defined below. It turns out that even when S is a safety property, so S^R is also safety property, R1 requires a liveness assumption about S. For a program with a fairness property F, we expect $S \wedge F$ to imply T. If the program describes only safety, then it can satisfy only safety properties, so it would be strange if we required S to satisfy a liveness property to use reduction. We will see that, to show S satisfies a safety property P, showing that S satisfies a possibility condition allows us to assume that T is satisfied.

To prove the theorem that asserts R1, we construct an abstract program $S \otimes S^R$, with variables $\mathbf{x}$ and $\mathbf{X}$, that describes the programs S and S^R running simultaneously in parallel. ($S \otimes S^R$ is just an identifier that names a formula, not an expression with formulas S and S^R and an operator $\otimes$. This unusual identifier is meant to remind us what the formula means.) We then show:

R1a. $\models S \;\Rightarrow\; \boldsymbol{\exists}\, \mathbf{X} : S \otimes S^R$

R1b. $\models S \otimes S^R \wedge T \;\Rightarrow\; S^R \wedge \Box I^R$

R1b implies $\models (\boldsymbol{\exists}\, \mathbf{X} : S \otimes S^R \wedge T) \Rightarrow (\boldsymbol{\exists}\, \mathbf{X} : S^R \wedge \Box I^R)$; and the variables $\mathbf{X}$ don't appear in T, so $\boldsymbol{\exists}\, \mathbf{X} : S \otimes S^R \wedge T$ equals $(\boldsymbol{\exists}\, \mathbf{X} : S \otimes S^R) \wedge T$. Therefore, R1a and R1b imply R1.

One property asserted by I^R is that, if the values of $\mathbf{x}$ describe a state in which none of the nonatomic operations of S being reduced is currently being executed, then $\mathbf{x} = \mathbf{X}$. The values of $\mathbf{x}$ and $\mathbf{X}$ are synchronized like this, despite the reduced program taking fewer steps than the original program, by having $S \otimes S^R$ take $\mathbf{X}$-stuttering steps—steps that leave the values of the variables $\mathbf{X}$ unchanged. R1b is satisfied because these $\mathbf{X}$-stuttering steps implement stuttering steps of S^R. R3 was satisfied in Lipton's original paper because he considered terminating programs and properties P that depend only on the initial and final states of the program. In those states, none of the nonatomic operations of S being reduced are being executed, so $\mathbf{x} = \mathbf{X}$. In what may have been the second paper published on reduction, Doeppner [10] satisfied R3 by proving only that an invariant is true when none of the reduced operations are being executed.

7.1.2 An Example

To explain reduction, we start by examining this commonly assumed rule: When reasoning about a multiprocess program in which interprocess com-

```
              ⋮
$r_1$:   R$_1$ ;
              ⋮
$r_k$:   R$_k$ ;
$c$:     C ;
$l_1$:   L$_1$ ;
              ⋮
$l_m$    L$_m$ ;
$o$:     ...
              ⋮
```

Figure 7.1: The nonatomic operation *RCL*.

munication is performed by atomic operations to shared data objects, the program can be represented with any grain of atomicity in which each atomic action accesses at most one shared data object.[1] The following is a statement of the rule in our science and the argument generally used to justify it.

Suppose S is a multiprocess program with a process that executes a nonatomic operation, which we call *RCL*, described by the statements shown in Figure 7.1. We assume this is "straight line" code, so an execution of *RCL* consists of $k+1+m$ steps. For now, we let S be the safety property described by the code; liveness is discussed later. We assume statements $\texttt{R}_i$ and $\texttt{L}_j$ can access only data local to the process, while statement `C` can also access shared data. The rule asserts that we can replace S with its reduced version S^R obtained by removing all the labels between (but not including) r_1 and o, so those $k+1+m$ statements are executed as a single step, and replacing the variables $\mathbf{x}$ with the variables $\mathbf{X}$. It is usually claimed that we can do this because other processes can't observe when the statements $\texttt{R}_i$ and $\texttt{L}_j$ are executed, so we can pretend that they are executed in the same step as statement `C`.

We can reduce other operations of the same form to atomic actions as well, reducing the operations one at a time. So, it suffices to see how it's done for just this single operation *RCL*, which may be executed multiple times.

[1]The rule was stated in print independently by Owicki and Gries [42] and by me [28], but I think it was well known at the time.

7.1.2.1 The Reduced Behaviors

Let R_i, C, and L_j be the TLA actions described by statements $\texttt{R}_i$, $\texttt{C}$, and $\texttt{L}_j$. An execution of *RCL* consists of a sequence of R_i steps, for $i \in 1\,..\,k$, followed by a C step, followed by a sequence of L_j steps, for $j \in 1\,..\,m$. In an execution of the operation during a behavior of the program, interleaved between those steps may be steps performed by other processes. (There can also be stuttering steps that leave the values of variables $\mathbf{x}$ unchanged, but they are irrelevant and we can ignore them.) A *reduced* execution of the operation execution is one in which all the steps of *RCL* occur one after the other, with no interleaved steps executed by other processes.

We transform a behavior to a *reduced* behavior, in which only reduced executions of *RCL* occur, by a procedure illustrated with the following portion of a behavior containing an execution of *RCL*, where *RCL* has one R_i statement ($k = 1$) and two L_j statements ($m = 2$). We show the name of the action that each step satisfies, where E_1, E_2, and E_3 are actions of other processes.

$$(7.1)\quad \cdots\ s_{41} \xrightarrow{R_1} s_{42} \xrightarrow{E_1} s_{43} \xrightarrow{C} s_{44} \xrightarrow{E_2} s_{45} \xrightarrow{E_3} s_{46} \xrightarrow{L_1} s_{47} \xrightarrow{L_2} s_{48}\ \cdots$$

Define $\mathcal{R}$ to be the state predicate that is true of a state iff that state occurs during an execution of *RCL* before the C action. In the part of a behavior shown in (7.1), $\mathcal{R}$ is true only in states s_{42} and s_{43}. Define $\mathcal{L}$ to be the state predicate asserting that the process is currently executing operation *RCL* after the C action, so in (7.1) it is true in states s_{44}–s_{47}. In general, if p is the process executing *RCL* of Figure 7.1, then $\mathcal{R}$ equals $pc(p) \in \{r2, \ldots, r_k, c\}$ and $\mathcal{L}$ equals $pc(p) \in \{l_1, \ldots, l_m\}$. The behavior is currently executing *RCL* iff the state predicate $\mathcal{R} \lor \mathcal{L}$ is true. Thus, the operation is not being executed iff $\neg(\mathcal{R} \lor \mathcal{L})$ is true.

Because R_i and L_j actions access only process-local state, they *commute* with actions from other processes, where two actions commute iff executing them in either order has the same effect. Recall that $A \cdot B$ is defined in Section 3.4.1.4 to be the action that is satisfied by a step $s \to t$ iff there is a state u such that $s \to u$ is an A step and $u \to t$ is a B step. Actions A and B commute iff $A \cdot B$ equals $B \cdot A$. If A and B commute, then for any states s, t, and u such that $s \xrightarrow{A} u \xrightarrow{B} t$, there exists a state v such that $s \xrightarrow{B} v \xrightarrow{A} t$. By commuting R_i and L_j actions with actions of other processes, moving R_i actions to the right and L_j actions to the left, we obtain a behavior in which every execution of *RCL* is reduced, with no steps of other processes interleaved. For example, commuting actions in this way converts the original behavior into the reduced behavior in which the portion of the original behavior shown in (7.1) is converted to:

$$(7.2) \quad \cdots\ s_{41} \xrightarrow{E_1} u_{42} \xrightarrow{R_1} s_{43} \xrightarrow{C} s_{44} \xrightarrow{L_1} u_{45} \xrightarrow{L_2} u_{46} \xrightarrow{E_2} u_{47} \xrightarrow{E_3} s_{48}\ \cdots$$

A state s_i is changed to a possibly different state u_i if commutativity was used to commute actions A and B in $\cdots \xrightarrow{A} s_i \xrightarrow{B} \cdots$. For example, state s_{46} had to be (possibly) changed twice to arrive at u_{46} because both the E_2 and E_3 actions had to be "moved across" state number 46 of the behavior to get from (7.1) to (7.2). If σ is the original behavior (7.1), we define $\Phi(\sigma)$ to be its reduced version (7.2), obtained by transforming all executions of *RCL* in this way.

Note that the states s_{41} and s_{48} are the same in σ and $\Phi(\sigma)$ because *RCL* is not being executed in those states. States s_{43} and s_{44}, which form the C step, are also the same in the original and reduced behaviors because the C action is not commuted with any action. The reduced behavior (7.2) is also a behavior of program S because it has the same initial state as (7.1), which satisfies *Init*, and every step satisfies a subaction of the next-state action *Next*.

7.1.2.2 The Program $S \otimes S^R$

We define $S \otimes S^R$ in terms of a mapping ϕ from states in the original behavior σ to states in the reduced behavior $\Phi(\sigma)$—in our example, from states of (7.1) to states of (7.2). First, we define $\phi(s)$ for all states except the ones in a C step, which in (7.1) are all states except for s_{43} and s_{44}. We call those two states "C states" and the other states "non-C states".

We transform a behavior containing executions of *RCL* to a reduced behavior by a sequence of action interchanges based on commutativity. There are two kinds of interchange, both involving an E_h action of a different process: one that moves an R_i step to the right of an E_h step, and one that moves an L_j step to the left of an E_h step. To go from behavior (7.1) to behavior (7.2) requires five interchanges. Figure 7.2 shows the sequence of behaviors created by performing these interchanges, where each behavior is obtained from the preceding one by interchanging the left-most R_i or L_j action that can be moved closer to the C action. The top behavior is the original behavior (7.1) and the bottom one is the reduced behavior (7.2). Observe that every state in one behavior equals the corresponding state in the next behavior except for the one state across which the actions are interchanged.

$$\begin{tikzcd}
\cdots & s_{41} \arrow[r, "R_1"] \arrow[d] & s_{42} \arrow[r, "E_1"] \arrow[dl, thick] & s_{43} \arrow[r, "C"] & s_{44} \arrow[r, "E_2"] & s_{45} \arrow[r, "E_3"] \arrow[d] & s_{46} \arrow[r, "L_1"] \arrow[d] & s_{47} \arrow[r, "L_2"] \arrow[d] & s_{48} \arrow[d] & \cdots \\
\cdots & s_{41} \arrow[r, "E_1"] \arrow[d] & u_{42} \arrow[r, "R_1"] \arrow[d] & s_{43} \arrow[r, "C"] & s_{44} \arrow[r, "E_2"] & s_{45} \arrow[r, "E_3"] \arrow[d] & s_{46} \arrow[r, "L_1"] \arrow[dr, thick] & s_{47} \arrow[r, "L_2"] \arrow[d] & s_{48} \arrow[d] & \cdots \\
\cdots & s_{41} \arrow[r, "E_1"] \arrow[d] & u_{42} \arrow[r, "R_1"] \arrow[d] & s_{43} \arrow[r, "C"] & s_{44} \arrow[r, "E_2"] & s_{45} \arrow[r, "L_1"] \arrow[dr, thick] & r_{46} \arrow[r, "E_3"] \arrow[d] & s_{47} \arrow[r, "L_2"] \arrow[d] & s_{48} \arrow[d] & \cdots \\
\cdots & s_{41} \arrow[r, "E_1"] \arrow[d] & u_{42} \arrow[r, "R_1"] \arrow[d] & s_{43} \arrow[r, "C"] & s_{44} \arrow[r, "L_1"] & u_{45} \arrow[r, "E_2"] \arrow[d] & r_{46} \arrow[r, "E_3"] \arrow[d] & s_{47} \arrow[r, "L_2"] \arrow[dr, thick] & s_{48} \arrow[d] & \cdots \\
\cdots & s_{41} \arrow[r, "E_1"] \arrow[d] & u_{42} \arrow[r, "R_1"] \arrow[d] & s_{43} \arrow[r, "C"] & s_{44} \arrow[r, "L_1"] & u_{45} \arrow[r, "E_2"] \arrow[d] & r_{46} \arrow[r, "L_2"] \arrow[dr, thick] & u_{47} \arrow[r, "E_3"] \arrow[d] & s_{48} \arrow[d] & \cdots \\
\cdots & s_{41} \arrow[r, "E_1"] & u_{42} \arrow[r, "R_1"] & s_{43} \arrow[r, "C"] & s_{44} \arrow[r, "L_1"] & u_{45} \arrow[r, "L_2"] & u_{46} \arrow[r, "E_2"] & u_{47} \arrow[r, "E_3"] & s_{48} & \cdots
\end{tikzcd}$$

Figure 7.2: Constructing (7.2) from (7.1).

The arrows in the picture are drawn according to the following rules. There is a (thin) downward pointing arrow from each non-C state that is unchanged by the interchange that yields the next behavior. From the one state in each behavior that is changed by the interchange, there is a (thick) diagonal arrow. If that state satisfies $\mathcal{R}$ (is before the C action), then the arrow points one state to the left of the changed state. If the state satisfies $\mathcal{L}$, then the arrow points one state to the right.

These arrows define a unique path from every non-C state s_i of the original behavior to a state in the reduced behavior. Define $\phi(s_i)$ to be that state in the reduced behavior. For the example in Figure 7.2, $\phi(s_{45}) = u_{47}$ because the sequence of states in the path from s_{45} in the top behavior to the bottom behavior is:

(7.3) $s_{45} \rightarrow s_{45} \rightarrow s_{45} \rightarrow r_{46} \rightarrow r_{46} \rightarrow u_{47}$

Figure 7.3 contains an arrow pointing from each non-C state s_i in the original behavior to the state $\phi(s_i)$ in the reduced behavior. Observe that for every non-C state s_i, the state $\phi(s_i)$ is a state in which operation *RCL* is not being executed—that is, a state satisfying $\neg(\mathcal{R} \vee \mathcal{L})$.

We define $\phi(s)$ for the C states so that if the C step is $s_i \xrightarrow{C} s_{i+1}$, then $\phi(s_i)$ is the first state to the left of s_i for which $\neg(\mathcal{R} \vee \mathcal{L})$ is true, and $\phi(s_{i+1})$ is the first state to the right of s_{i+1} for which $\neg(\mathcal{R} \vee \mathcal{L})$ is true. In other words, $\phi(s_i)$ and $\phi(s_{i+1})$ are the states of the reduced behavior in which the execution of *RCL* begins and ends, respectively. The complete mapping ϕ

$$\begin{tikzcd}
\cdots & s_{41} \arrow[r, "R_1"] \arrow[d] & s_{42} \arrow[r, "E_1"] \arrow[dl] & s_{43} \arrow[r, "C"] & s_{44} \arrow[r, "E_2"] & s_{45} \arrow[r, "E_3"] \arrow[drr] & s_{46} \arrow[r, "L_1"] \arrow[drr] & s_{47} \arrow[r, "L_2"] \arrow[dr] & s_{48} \arrow[d] & \cdots \\
\cdots & s_{41} \arrow[r, "E_1"] & u_{42} \arrow[r, "R_1"] & s_{43} \arrow[r, "C"] & s_{44} \arrow[r, "L_1"] & u_{45} \arrow[r, "L_2"] & u_{46} \arrow[r, "E_2"] & u_{47} \arrow[r, "E_3"] & s_{48} & \cdots
\end{tikzcd}$$

Figure 7.3: The mapping $s_i \rightarrow \phi(s_i)$ for non-C states.

$$\begin{tikzcd}
\cdots & s_{41} \arrow[r, gray, "R_1"] \arrow[d, gray] & s_{42} \arrow[r, gray, "E_1"] \arrow[dl, gray] & s_{43} \arrow[r, "C"] \arrow[dl] & s_{44} \arrow[r, gray, "E_2"] \arrow[drr] & s_{45} \arrow[r, gray, "E_3"] \arrow[drr, gray] & s_{46} \arrow[r, gray, "L_1"] \arrow[drr, gray] & s_{47} \arrow[r, gray, "L_2"] \arrow[dr, gray] & s_{48} \arrow[d, gray] & \cdots \\
\cdots & s_{41} \arrow[r, gray, "E_1"] & u_{42} \arrow[r, "R_1"] & s_{43} \arrow[r, "C"] & s_{44} \arrow[r, "L_1"] & u_{45} \arrow[r, "L_2"] & u_{46} \arrow[r, gray, "E_2"] & u_{47} \arrow[r, gray, "E_3"] & s_{48} & \cdots
\end{tikzcd}$$

Figure 7.4: The complete mapping $s_i \rightarrow \phi(s_i)$.

for our example is shown in Figure 7.4, where the mapping for non-C states is in gray.

Formula $S \otimes S^R$ is defined so that it is satisfied by a behavior σ iff σ satisfies S (which describes the values of variables $\mathbf{x}$) and the values of the variables $\mathbf{X}$ in any state s_k of σ equal the values of the variables $\mathbf{x}$ in the state $\phi(s_k)$ of the reduced behavior $\Phi(\sigma)$. Since this assigns values of variables $\mathbf{X}$ to every state of every behavior σ so that the behavior satisfies $S \otimes S^R$, we see that $\models S \Rightarrow \exists\, \mathbf{X} : S \otimes S^R$ is true, so R1a is satisfied.

From Figure 7.4, we see that for any k:

ϕ1. If $s_k \rightarrow s_{k+1}$ is an E_h step (so E_h is an action of another process) then $\phi(s_k) \rightarrow \phi(s_{k+1})$ is also an E_h step.

ϕ2. If $s_k \rightarrow s_{k+1}$ is an R_i or L_j step, then $\phi(s_k) = \phi(s_{k+1})$.

ϕ3. If $s_k \rightarrow s_{k+1}$ is a C step, then $\phi(s_k)$ and $\phi(s_{k+1})$ are the first and last states of an execution of operation *RCL* in the reduced behavior $\Phi(\sigma)$, which is an execution with no interleaved steps of other process actions.

Recall that $\overline{G}$ equals G WITH $\mathbf{x} \leftarrow \mathbf{X}$ for any formula G. A step of S^R is either an $\overline{E_h}$ step, a $\overline{D}$ step, where D is an action that performs an execution of operation *RCL* as a single step, or a stuttering step that leaves the variables $\mathbf{X}$ unchanged. From ϕ1–ϕ3, we see that if behavior (7.1) satisfies $S \otimes S^R$ (as well as S), then each step of that behavior is either an $\overline{E_h}$ step (by ϕ1), a $\overline{D}$ step (by ϕ3) or leaves the variables $\mathbf{X}$ unchanged (by ϕ2). Hence each step of $S \otimes S^R$ satisfies the next-state action of S^R.

The initial-state predicate of S^R is $\overline{Init}$ (remember that *Init* is the initial-state predicate of S). Since operation *RCL* is not being executed in the initial state s_0 of (7.1), $\phi(s_0)$ equals s_0, which implies that s_0 satisfies $\overline{Init}$. Thus, if (7.1) satisfies $S \otimes S^R$, then it satisfies S^R. Therefore, $S \otimes S^R \Rightarrow S^R$ is satisfied—or so it seems.

The reasoning works in this example because the behavior (7.1) contains a complete execution of operation *RCL*. However, S is a safety property, which means that the process can stop executing actions at any point during the execution of the operation. The general definition of the reduced version of a behavior, which includes a possibly incomplete execution of *RCL*, is that actions performing steps of an *RCL* execution are made to occur together by commuting R_i actions to the right, L_j actions to the left, and leaving a C action unmoved. (If there is no C step, then the last R_i action can be left unmoved.) If s is the last state of a C step, $\phi(s)$ is defined to be the state after the last step of the *RCL* operation being executed. That state will satisfy $\neg(\mathcal{R} \vee \mathcal{L})$ iff the behavior contains a complete execution of the operation.

The one part of what we've done that's not correct in the presence of a partial execution is $\phi 3$. Statement $\phi 3$ is vacuously true if the partial execution doesn't contain a C step. In that case there are only R_i steps which correspond to stuttering steps of S^R, so the behavior of $S \otimes S^R$ is a behavior of S^R in which that *RCL* action doesn't occur. The problem in $\phi 3$ arises in our example if the execution contains a C step but doesn't contain both the L_1 and L_2 steps.

The solution is to rule out such behaviors. That's what the hypothesis T in R1 and R1b does. Formula T must assert that any execution of *RCL* that includes the C step must complete. The operation has performed the C step but has not completed iff $\mathcal{L}$ is true. So we want to allow only behaviors in which it is not the case that $\mathcal{L}$ eventually becomes true and remains true forever. Such behaviors are ones satisfying $\neg\Diamond\Box\mathcal{L}$, which is equivalent to $\Box\Diamond\neg\mathcal{L}$. So, we can restate R1b as this assumption:

$$\models S \otimes S^R \wedge (\Box\Diamond\neg\mathcal{L}) \;\Rightarrow\; S^R \wedge \Box I^R$$

The argument showing $S \otimes S^R$ implies S^R for the behavior (7.1) applies to all behaviors of the program of Figure 7.1 satisfying $\Box\Diamond\neg\mathcal{L}$. That is, we have shown:

$$\models S \otimes S^R \wedge (\Box\Diamond\neg\mathcal{L}) \;\Rightarrow\; S^R$$

is satisfied by this program S. To complete the proof of R1b, we must define I^R and show that it is an invariant of $S \otimes S^R$. Since we have seen that S satisfies R1a, this will show that it satisfies R1 with T equal to $\Box\Diamond\neg\mathcal{L}$. The assumption $\Box\Diamond\neg\mathcal{L}$ is discussed in Section 7.1.4.

7.1.2.3 The Invariant I^R

The invariant I^R of $S \otimes S^R$ that relates the values of the variables $\mathbf{x}$ to those of the variables $\mathbf{X}$ follows from three additional properties of the mapping ϕ. We can see why those properties hold from Figures 7.2–7.4.

The most obvious of these properties follows from the observation that if $\neg(\mathcal{R} \vee \mathcal{L})$ is true in a state s, then only downward pointing arrows are drawn from it in the figures. This shows:

$\phi 4$. For any state s in which $\neg(\mathcal{R} \vee \mathcal{L})$ is true, $\phi(s) = s$.

For the next property, look at the path (7.3) from the state s_{45} to the state $\phi(s_{45})$, which equals u_{47}. Follow that path in Figure 7.2. Observe that each step in the path either leaves the state unchanged (is a stuttering step) or else is an L_1 or L_2 step. (To see this, look at the horizontal arrows just above the diagonal arrows.) We can see from the figure that this is true of the path from state s to state $\phi(s)$ for the states s_{46} and s_{47} as well. The rule for drawing the arrows implies that this is true in general, for all executions of the program of Figure 7.1. The path from every non-C state s for which $\mathcal{L}$ is true to $\phi(s)$ consists of a sequence of steps each of which is either a stuttering step or an L_j step for some j. From Figure 7.4, it's clear that this is also true for the C state for which $\mathcal{L}$ is true.

Let's define L to equal $L_1 \vee \ldots \vee L_m$, so any L_j step is an L step. We have seen that we can get from a state s in which $\mathcal{L}$ is true to the state $\phi(s)$ by a sequence of stuttering steps and/or L steps. Recall that in Section 4.3.2, for any action A we defined A^+ to equal $A \vee (A \cdot A) \vee (A \cdot A \cdot A) \vee \ldots$. Therefore, a step $s \to t$ satisfies $([L]_{\langle \mathbf{x} \rangle})^+$, which we abbreviate $[L]^+_{\langle \mathbf{x} \rangle}$, iff we can get from state s to state t by a sequence of L steps or steps that leave the variables $\mathbf{x}$ unchanged. Let's write the subscript $\langle \mathbf{x} \rangle$ as simply $\mathbf{x}$, so $[A]_{\mathbf{x}}$ and $\langle A \rangle_{\mathbf{x}}$ mean $[A]_{\langle \mathbf{x} \rangle}$ and $\langle A \rangle_{\langle \mathbf{x} \rangle}$ for any action A.

We have thus seen:

$\phi 5$. For any state s in which $\mathcal{L}$ is true, $s \to \phi(s)$ is an $[L]^+_{\mathbf{x}}$ step.

Following the path from s to $\phi(s)$ backwards for states s in which $\mathcal{R}$ is true similarly leads to the following, where R equals $R_1 \vee \ldots \vee R_k$.[2]

$\phi 6$. For any state s in which $\mathcal{R}$ is true, $\phi(s) \to s$ is an $[R]^+_{\mathbf{x}}$ step.

[2]The action R bears no relation to the superscript R in S^R. It is traditional to name the action R for *R*ight-mover because of the way the reduced behavior is constructed; and there seems to be no better superscript than R to signify *reduced*.

Finally, Figure 7.4 shows that for any state s of the original behavior, $\phi(s)$ is always a state in the reduced behavior in which the *RCL* operation is not being executed, so $\neg(\mathcal{R} \vee \mathcal{L})$ is true for $\phi(s)$. Because the rule for drawing the arrows in Figure 7.2 creates a leftward pointing arrow whenever an R step is moved to the right and a rightward pointing arrow whenever an L step is moved to the left, this is true in general. Therefore, we have:

$\phi7$. For any state s, the state predicate $\neg(\mathcal{R} \vee \mathcal{L})$ is true in state $\phi(s)$.

Statements $\phi4$–$\phi7$ give us relations between s and $\phi(s)$ for all states s of a behavior of our example program. We now have to turn them into relations between the values of the variables $\mathbf{x}$ and the variables $\mathbf{X}$ in any state s in a behavior of $S \otimes S^R$.

It's easy to do this for $\phi4$. The values of the variables $\mathbf{X}$ in state s are the values of $\mathbf{x}$ in $\phi(s)$. Since $\phi(s) = s$ if s satisfies $\neg(\mathcal{R} \vee \mathcal{L})$, this means that $\mathbf{x} = \mathbf{X}$ is true for any reachable state of $S \otimes S^R$ satisfying $\neg(\mathcal{R} \vee \mathcal{L})$. In other words:

(7.4) $\neg(\mathcal{R} \vee \mathcal{L}) \Rightarrow (\mathbf{X} = \mathbf{x})$ is an invariant of $S \otimes S^R$

To express the relations between $\mathbf{x}$ and $\mathbf{X}$ implied by $\phi5$ and $\phi6$, we need a bit of notation. For any action A containing only the variables $\mathbf{x}$, a step $s \rightarrow t$ satisfies A iff formula A is true when we substitute for the variables $\mathbf{x}$ their values in state s and for $\mathbf{x}'$ their values in state t. Therefore, $\phi(s) \rightarrow s$ is an $[R]^+_{\mathbf{x}}$ step iff $[R]^+_{\mathbf{x}}$ is true when we substitute the values of $\mathbf{x}$ in $\phi(s)$ for the variables $\mathbf{x}$ and the values of $\mathbf{x}$ in s for the primed variables $\mathbf{x}'$. But the values of $\mathbf{x}$ in $\phi(s)$ are by definition the values of $\mathbf{X}$ in state s. So $\phi(s) \rightarrow s$ is an $[R]^+_{\mathbf{x}}$ step iff s satisfies the formula we get by substituting $\mathbf{X}$ for $\mathbf{x}$ and $\mathbf{x}$ for $\mathbf{x}'$ in $[R]^+_{\mathbf{x}}$. Using the construct AWITH defined in Section 5.4.4.1, we write the formula produced by these substitutions as:

$$[R]^+_{\mathbf{x}} \text{ AWITH } \mathbf{x} \leftarrow \mathbf{X},\, \mathbf{x}' \leftarrow \mathbf{x}$$

so formula $\phi6$ implies:

(7.5) $\mathcal{R} \Rightarrow ([R]^+_{\mathbf{x}} \text{ AWITH } \mathbf{x} \leftarrow \mathbf{X},\, \mathbf{x}' \leftarrow \mathbf{x})$ is an invariant of $S \otimes S^R$

Remember that in this formula, the definition of $[R]^+_{\mathbf{x}}$ must be fully expanded before the AWITH substitution is performed.

We can obtain a similar relation between $\mathbf{x}$ and $\mathbf{X}$ from $\phi5$. The statement that $s \rightarrow \phi(s)$ is an $[L]^+_{\mathbf{x}}$ step is equivalent to the statement that state s is satisfied by the formula obtained from $[L]^+_{\mathbf{x}}$ by substituting for variables $\mathbf{x}$ their values in state s and substituting for $\mathbf{x}'$ the values of $\mathbf{x}$ in state

$\phi(s)$, the latter being the values of $\mathbf{X}$ in state s. Therefore, $\phi5$ implies the following, where $\mathbf{x} \leftarrow \mathbf{x}$ has been eliminated from the AWITH formula, since it just states that $\mathbf{x}$ is substituted for itself:

(7.6) $\mathcal{L} \Rightarrow ([L]^{+}_{\mathbf{x}}$ AWITH $\mathbf{x}' \leftarrow \mathbf{X})$ is an invariant of $S \otimes S^{R}$

Finally, since the values of $\mathbf{X}$ in state s equal the values of $\mathbf{x}$ in state $\phi(s)$, from $\phi7$ we obtain:

(7.7) $\neg(\mathcal{R} \vee \mathcal{L})$ WITH $\mathbf{x} \leftarrow \mathbf{X}$ is an invariant of $S \otimes S^{R}$

The invariant I^{R} of $S \otimes S^{R}$ relating the values of $\mathbf{x}$ and $\mathbf{X}$ is the conjunction of the invariants (7.4)–(7.7):

$$\begin{aligned} I^{R} \;\triangleq\; & \wedge\ \neg(\mathcal{R} \vee \mathcal{L}) \;\Rightarrow\; (\mathbf{X} = \mathbf{x}) \\ & \wedge\ \mathcal{R} \;\Rightarrow\; ([R]^{+}_{\mathbf{x}} \text{ AWITH } \mathbf{x} \leftarrow \mathbf{X}, \mathbf{x}' \leftarrow \mathbf{x}) \\ & \wedge\ \mathcal{L} \;\Rightarrow\; ([L]^{+}_{\mathbf{x}} \text{ AWITH } \mathbf{x}' \leftarrow \mathbf{X}) \\ & \wedge\ \neg(\mathcal{R} \vee \mathcal{L}) \text{ WITH } \mathbf{x} \leftarrow \mathbf{X} \end{aligned}$$

7.1.3 Reduction In General

So far, we have been considering reduction of an operation *RCL* of one process of a multiprocess program having the form shown in Figure 7.1. We now extend this to a more general situation. First, some observations about reducing *RCL*.

The invariant I^{R} is described in terms of the actions R and L, which are the disjunction of the actions R_i and L_j, respectively. The fact that R and L were defined in this particular way was irrelevant. All we required was that an execution of the operation *RCL* consists of a sequence of R steps followed by a C step followed by a sequence of L steps. We didn't need actions E_h of other processes to commute with each of the actions R_i and L_j. In drawing the diagrams illustrated by Figures 7.2 and 7.4, we didn't have to identify the subactions of R and L that are involved. We could have replaced each R_i by R and each L_i by L. We only needed to assume that each action of another process commutes with R and L, not with each R_i and L_j individually. (For example, an $E_h \cdot L_1$ step could be an $L_2 \cdot E_h$ step.) Similarly, we didn't have to require that R and L commute with particular actions of other processes. We just required R and L to commute with the disjunction of the next-state relations of all the other processes—an action we will call E.

Furthermore, we didn't need to require that R and L commute with E. To move R steps to the right, we just required that for any pair of steps

$s \xrightarrow{R} u \xrightarrow{E} t$ there is a pair of steps $s \xrightarrow{E} v \xrightarrow{R} t$ for some state v. This condition can be stated as the requirement $R \cdot E \Rightarrow E \cdot R$. When this formula holds, we say that R *right-commutes* with E, and that E *left-commutes* with R. Similarly, to move L steps to the left, we don't need $E \cdot L$ to equal $L \cdot E$; we need only require $E \cdot L \Rightarrow L \cdot E$, which asserts that L left-commutes with E (and E right-commutes with L).

Finally, suppose that A is another action of the same process containing the *RCL* operation, so A does not allow steps that implement the *RCL* operation. Since the next step of the process after an R step can only be an R step or a C step, an A step cannot follow an R step. This implies that $R \cdot A$ must equal FALSE, which means that $R \cdot A \Rightarrow A \cdot R$ is trivially true. Similarly, the only step of the process that can immediately precede an L step is an L step or a C step. Therefore, $A \cdot L$ must equal FALSE, so $A \cdot L \Rightarrow L \cdot A$ must be true. Therefore, other actions of the process containing *RCL* satisfy the same commutativity requirements as actions of other processes. This implies that we can completely forget about processes. We just assume that the program's next-state action equals $E \vee R \vee C \vee L$, where R right-commutes with E and L left-commutes with E. If we represent the program as a collection of processes, there is no need for R, C, and L steps to all be steps of the same process.

What we need to require is that execution of an *RCL* operation consists of a sequence of R steps followed by a C step followed by a sequence of L steps. To express this requirement, we generalize $\mathcal{R}$ and $\mathcal{L}$ from assertions about a process's control state to arbitrary state predicates satisfying certain conditions. We take as primitives the state predicates $\mathcal{R}$ and $\mathcal{L}$ and the actions E and M such that the program's next state relation equals $E \vee M$, where M describes the operation to be reduced. The actions R, C, and L will be defined in terms of $\mathcal{R}$, $\mathcal{L}$, and M. We therefore assume the original program S is defined by:

$$S \;\triangleq\; \mathit{Init} \wedge \Box[E \vee M]_{\mathbf{x}}$$

As in our example, we assume that $\mathcal{R}$ is true when execution of the operation described by M is in its first phase, where execution has begun but the C action has not yet been executed; and $\mathcal{L}$ is true when execution is in its second phase, where the C action has been executed but the operation execution has not yet terminated. This is implied by the following assumptions on executions of an operation described by the action M:

M1. In the initial state, M is not in the middle of an execution, expressed by $\mathit{Init} \Rightarrow \neg(\mathcal{R} \vee \mathcal{L})$.

M2. An E step can't change the current phase of an execution of M, expressed by $E \Rightarrow (\mathcal{R}' \equiv \mathcal{R}) \land (\mathcal{L}' \equiv \mathcal{L})$.

M3. An M step can't go from the second phase to the first phase, expressed by $\neg(\mathcal{L} \land M \land \mathcal{R}')$.

M4. The two phases are disjoint, expressed by $\neg(\mathcal{R} \land \mathcal{L})$.

We can define the actions R, L, and C in terms of M, $\mathcal{R}$, and $\mathcal{L}$ by:

$$R \;\triangleq\; M \land \mathcal{R}' \qquad L \;\triangleq\; \mathcal{L} \land M \qquad C \;\triangleq\; (\neg\mathcal{L}) \land M \land (\neg\mathcal{R}')$$

A complete execution of the operation described by M consists of a sequence of M steps beginning and ending in a state in which the M operation is not being executed—in other words, in a state satisfying $\neg(\mathcal{R} \lor \mathcal{L})$. Therefore, the action M^R that executes the complete operation as a single step for the variables $\mathbf{X}$ is:

$$M^R \;\triangleq\; (\neg(\mathcal{R} \lor \mathcal{L}) \land M^+ \land \neg(\mathcal{R} \lor \mathcal{L})') \text{ WITH } \mathbf{x} \leftarrow \mathbf{X}$$

We can therefore define S^R as follows, where $Init^R$ and E^R are $Init$ and E with the variables $\mathbf{x}$ replaced by the variables $\mathbf{X}$:

$$S^R \;\triangleq\; Init^R \land \Box[E^R \lor M^R]_{\mathbf{X}}$$

If $\mathcal{R}$ always equals FALSE, then R equals FALSE so there is no R action. Similarly, there is no L action if $\mathcal{L}$ always equals FALSE. If there is neither an R nor an L action, then M equals C and S^R equals (S WITH $\mathbf{x} \leftarrow \mathbf{X}$) so reduction accomplishes nothing.

Ignoring liveness, reduction is described in TLA by a theorem asserting that R1 (with T equal to $\Box\Diamond\neg\mathcal{L}$) is implied by the definitions of S, R, L, M, and S^R above, assumptions M1–M4, and the commutativity relations assumed of R, L, and E.

M1–M4 and the commutativity relations are action formulas. (Remember that a state predicate is an action whose value depends only on the first state of a step.) Those action formulas don't have to be true of all possible steps, just on steps in behaviors satisfying S. (The action formulas are usually meaningless for states not satisfying some type-correctness predicate.) If $\mathbf{x}$ is the list of all variables that can appear in S and A, then the assertion that an action formula A is true for all steps of behaviors satisfying S can be written as $\models S \Rightarrow \Box[A]_{\mathbf{x}}$. Here is reduction, without liveness, expressed as a theorem.

Theorem 7.1 Assume *Init*, $\mathcal{L}$, and $\mathcal{R}$ are state predicates, M and E are actions, $\mathbf{x}$ is a list of all variables appearing in these formulas, and $\mathbf{X}$ is a list of the same number of variables different from the variables $\mathbf{x}$. Define

$$\begin{array}{l}
S \triangleq Init \wedge \Box[E \vee M]_{\mathbf{x}} \qquad R \triangleq M \wedge \mathcal{R}' \qquad L \triangleq \mathcal{L} \wedge M \\
Init^R \triangleq Init \text{ WITH } \mathbf{x} \leftarrow \mathbf{X} \qquad E^R \triangleq E \text{ WITH } \mathbf{x} \leftarrow \mathbf{X} \\
M^R \triangleq (\neg(\mathcal{R} \vee \mathcal{L}) \wedge M^+ \wedge \neg(\mathcal{R} \vee \mathcal{L})') \text{ WITH } \mathbf{x} \leftarrow \mathbf{X} \\
S^R \triangleq Init^R \wedge \Box[E^R \vee M^R]_{\mathbf{X}} \\
I^R \triangleq \begin{array}[t]{l}
\wedge\ \neg(\mathcal{R} \vee \mathcal{L}) \Rightarrow (\mathbf{X} = \mathbf{x}) \\
\wedge\ \mathcal{R} \Rightarrow ([R]^+_{\mathbf{x}} \text{ AWITH } \mathbf{x} \leftarrow \mathbf{X},\, \mathbf{x}' \leftarrow \mathbf{x}) \\
\wedge\ \mathcal{L} \Rightarrow ([L]^+_{\mathbf{x}} \text{ AWITH } \mathbf{x}' \leftarrow \mathbf{X}) \\
\wedge\ \neg(\mathcal{R} \vee \mathcal{L}) \text{ WITH } \mathbf{x} \leftarrow \mathbf{X}
\end{array}
\end{array}$$

and assume:

$$\begin{array}{ll}
(1) & \models Init \Rightarrow \neg(\mathcal{R} \vee \mathcal{L}) \\
(2) & \models S \Rightarrow \Box\, [\begin{array}[t]{l}
\wedge\ E \Rightarrow (\mathcal{R}' \equiv \mathcal{R}) \wedge (\mathcal{L}' \equiv \mathcal{L}) \\
\wedge\ \neg(\mathcal{L} \wedge M \wedge \mathcal{R}') \\
\wedge\ \neg(\mathcal{R} \wedge \mathcal{L}) \\
\wedge\ R \cdot E \Rightarrow E \cdot R \\
\wedge\ E \cdot L \Rightarrow L \cdot E\ ]_{\mathbf{x}}
\end{array}
\end{array}$$

Then $\models S \wedge \Box\Diamond\neg\mathcal{L} \Rightarrow \exists\!\!\exists\, \mathbf{X} : S^R \wedge \Box I^R$.

Assumption (1) and the first three conjuncts in the action of assumption (2) are the conditions M1–M4, which assert that an execution of the operation described by the action M consists of a sequence of L steps followed by a C step followed by a sequence of R steps. The final two conjuncts in the action of assumption (2) are the assumptions that R right-commutes with E and L left-commutes with E.

In practice, R, L, and E will be defined to be the disjunction of subactions. This allows us to decompose the proofs of those commutativity conditions by using the following theorem that is proved in the Appendix.

Theorem 7.2 If $A \equiv \exists\, i \in I : A_i$ and $B \equiv \exists\, j \in J : B_j$ for actions A_i and B_j, then:

$$\models (\forall\, i \in I,\, j \in J : A_i \cdot B_j \Rightarrow B_j \cdot A_i) \Rightarrow (A \cdot B \Rightarrow B \cdot A)$$

By this theorem, to show that R right-commutes with E, it suffices to show that each subaction in the definition of R right-commutes with each subaction in the definition of E. Similarly, L left-commutes with E if each subaction of L left-commutes with each subaction of E.

7.1.4 The Hypothesis $\Box\Diamond\neg\mathcal{L}$

Formula $\Box\Diamond\neg\mathcal{L}$ is a liveness property. When we add liveness conditions to S and hope to use reduction to prove that it satisfies a liveness property P, we would expect S to imply $\Box\Diamond\neg\mathcal{L}$. But we wouldn't expect to have to add a liveness property to S in order to verify that it satisfies a safety property. In fact, to verify a safety property, we don't have to prove that $\neg\mathcal{L}$ is always eventually true. The following theorem implies that, to verify S satisfies a safety property, we can simply assume that $\Box\Diamond\neg\mathcal{L}$ is true if it is always *possible* for $\neg\mathcal{L}$ to eventually become true. To understand the theorem, recall that Section 4.3.2 defines what it means for it to be always possible for a state predicate Q to eventually become true—namely, that the program $\mathit{Init} \wedge \Box[\mathit{Next}]_{\mathbf{x}}$ implies $\Box\,\mathbb{E}([\mathit{Next}]_{\mathbf{x}}^{+} \wedge Q')$. (See (4.26).) This theorem is proved in the Appendix:

Theorem 7.3 If S equals $\mathit{Init} \wedge \Box[\mathit{Next}]_{\mathbf{x}}$, P is a safety property, and Q is a state predicate such that $\models S \Rightarrow \Box\,\mathbb{E}([\mathit{Next}]_{\mathbf{x}}^{+} \wedge Q')$, then $\models S \wedge \Box\Diamond Q \Rightarrow P$ implies $\models S \Rightarrow P$.

The theorem allows us to replace the formula T in R1b (which we later defined to equal $\Box\Diamond\neg\mathcal{L}$) with $\Box\,\mathbb{E}([\mathit{Next}]_{\mathbf{x}}^{+} \wedge \neg\mathcal{L}')$. Section 4.3.2 explains that we can verify this possibility condition by finding a fairness condition F for S and verifying $\models S \wedge F \Rightarrow \Box\Diamond\neg\mathcal{L}$. This is the only case I know of in which a possibility condition is used to verify a correctness property.

7.1.5 Adding Liveness

Theorem 7.1 allows us to deduce safety properties of the program S from safety properties of the coarser-grain program S^R. We also want to deduce liveness properties of S from liveness properties of S^R. We deduce liveness properties of a program from fairness properties of program actions. To deduce liveness properties of S by proving liveness properties of S^R, we extend Theorem 7.1 so its conclusion is

(7.8) $\models S \wedge F \wedge \Box\Diamond\neg\mathcal{L} \;\Rightarrow\; \boldsymbol{\exists}\, \mathbf{X} : S^R \wedge \Box I^R \wedge F^R$

where F is the conjunction of fairness properties of subactions of the next-state action $E \vee M$ of S and F^R is the conjunction of fairness properties of subactions of the next-state action $E^R \vee M^R$ of S^R.

To understand how this is done, it helps to think in terms of the program $S \otimes S^R$, a behavior of which is a behavior both of S (described by the values of variables $\mathbf{x}$) and of S^R (described by the values of variables $\mathbf{X}$) whose existence is asserted by (7.8). We expect S to satisfy (7.8) because $S \otimes S^R$ satisfies:

$$(7.9) \quad \models S \otimes S^R \land F \land \Box\Diamond\neg\mathcal{L} \;\Rightarrow\; S^R \land \Box I^R \land F^R$$

Formula F^R should assert fairness of subactions A^R of $E^R \lor M^R$. For simplicity, we consider only actions A^R that are subactions of either E^R or M^R, which I expect will usually be the case. When it is not the case, the requirements for deducing fairness of A^R include requirements for deducing fairness of both $A^R \land E^R$ and $A^R \land M^R$ [8].

7.1.5.1 Fairness of Subactions of E^R

We defined S^R so that E^R equals $\overline{E}$. (Recall that we defined $\overline{G}$ to equal G WITH $\mathbf{x} \leftarrow \mathbf{X}$ for any formula G.) Property $\phi 1$ of Section 7.1.2.2 for our example generalizes to show that, in any behavior of $S \otimes S^R$, an E step is also an $\overline{E}$ step.

Because each E^R step corresponds to a single E step in a behavior of $S \otimes S^R$, we expect fairness of a subaction A^R of E^R to be implied by fairness of a single subaction A of E. In other words, we expect this to be true:

$$(7.10) \quad \models S \otimes S^R \land \mathrm{XF}_{\mathbf{x}}(A) \;\Rightarrow\; \mathrm{XF}_{\mathbf{X}}(A^R)$$

where XF is either WF or SF. By (4.14) and (4.22) we have

$$\begin{aligned} \mathrm{XF}_{\mathbf{x}}(A) &\equiv (⛋⛋\, \mathbb{E}\langle A \rangle_{\mathbf{x}} \Rightarrow \Box\Diamond\langle A \rangle_{\mathbf{x}}) \\ \mathrm{XF}_{\mathbf{X}}(A^R) &\equiv (⛋⛋\, \mathbb{E}\langle A^R \rangle_{\mathbf{X}} \Rightarrow \Box\Diamond\langle A^R \rangle_{\mathbf{X}}) \end{aligned}$$

where $⛋⛋$ means $\Diamond\Box$ if XF is WF and $\Box\Diamond$ if XF is SF. These formulas and a little temporal logic imply that to prove (7.10) it suffices to prove these two theorems:

$$(7.11) \quad \models S \otimes S^R \;\Rightarrow\; (\Box\Diamond\langle A \rangle_{\mathbf{x}} \Rightarrow \Box\Diamond\langle A^R \rangle_{\mathbf{X}})$$

$$(7.12) \quad \models S \otimes S^R \;\Rightarrow\; \Box(\mathbb{E}\langle A^R \rangle_{\mathbf{X}} \Rightarrow \mathbb{E}\langle A \rangle_{\mathbf{x}})$$

We make (7.11) true by requiring every $\langle A \rangle_{\mathbf{x}}$ step in a behavior of $S \otimes S^R$ to be an $\langle A^R \rangle_{\mathbf{X}}$ step. Every E step in a behavior of $S \otimes S^R$ is an E^R step because E^R equals $\overline{E}$. This suggests that we want A^R to equal $\overline{A}$. (Note that $\overline{\langle A \rangle_{\mathbf{x}}}$ equals $\langle \overline{A} \rangle_{\mathbf{X}}$ because $\overline{\mathbf{x}}$ equals $\mathbf{X}$.)

Every E step in a behavior of $S \otimes S^R$ is an E^R step because of property $\phi 1$, and that property holds because of the commutativity assumptions of action E with respect to actions R and L. Therefore, we must require that $\langle A\rangle_{\mathbf{x}}$ satisfies the same commutativity relation as E, namely:

$$(7.13)\quad \models S \;\Rightarrow\; \Box[\,(R \cdot \langle A\rangle_{\mathbf{x}} \Rightarrow \langle A\rangle_{\mathbf{x}} \cdot R) \;\wedge\; (\langle A\rangle_{\mathbf{x}} \cdot L \Rightarrow L \cdot \langle A\rangle_{\mathbf{x}})\,]_{\mathbf{x}}$$

With this assumption, defining A^R to equal $\overline{A}$ will make (7.11) true.

The commutativity relations satisfied by E were used to define the mapping ϕ, which in turn was used to define S^R. For example, in Figures 7.2–7.4, the state u_{42}, which equals $\phi(s_{43})$, was chosen when right-commuting R_1 with E_1 so that $s_{41} \rightarrow u_{42}$ is an E_1 step and $u_{42} \rightarrow s_{43}$ is an R_1 step. Suppose that A is a subaction of E_1 and $s_{41} \rightarrow s_{42}$ is an $\langle A\rangle_{\mathbf{x}}$ step. Nothing in the construction in those figures, which describes the choice of ϕ used to define S^R in Theorem 7.1, ensures that the E_1 step $s_{41} \rightarrow u_{42}$ is also an $\langle A\rangle_{\mathbf{x}}$ step. However, the assumption (7.13) ensures that we could have chosen u_{42} to make $s_{41} \rightarrow u_{43}$ an $\langle A\rangle_{\mathbf{x}}$ step.

In general, we can use (7.13) to ensure that every E step in the original behavior that is an $\langle A\rangle_{\mathbf{x}}$ step remains an $\langle A\rangle_{\mathbf{x}}$ step whenever that step is commuted with an R or L step, so it remains an $\langle A\rangle_{\mathbf{x}}$ step in the reduced behavior. We can therefore define S^R so that (7.12) is true, so this is not a problem when we want to ensure that (7.10) holds for a single action A. However, we may want (7.10) to hold for multiple subactions A of E, and a single E step can be an $\langle A\rangle_{\mathbf{x}}$ step for more than one of those subactions A. We will return to this problem in Section 7.1.5.3. Now, we consider making (7.12) true.

We can deduce (7.12) from these two theorems:

$$\begin{array}{lll}(7.14) & (a) & \models S \otimes S^R \;\Rightarrow\; \Box(\mathbb{E}\langle A^R\rangle_{\mathbf{X}} \;\Rightarrow\; \overline{\mathbb{E}\langle A\rangle_{\mathbf{x}}}\,) \\ & (b) & \models S \otimes S^R \;\Rightarrow\; \Box(\,\overline{\mathbb{E}\langle A\rangle_{\mathbf{x}}} \;\Rightarrow\; \mathbb{E}\langle A\rangle_{\mathbf{x}})\end{array}$$

Since $\langle A^R\rangle_{\mathbf{X}}$ equals $\overline{\langle A\rangle_{\mathbf{x}}}$ and formulas $\mathbb{E}\overline{\langle A\rangle_{\mathbf{x}}}$ and $\overline{\mathbb{E}\langle A\rangle_{\mathbf{x}}}$ contain only the variables $\mathbf{X}$, (7.14a) is equivalent to:

$$(7.15)\quad \models S^R \;\Rightarrow\; \Box(\mathbb{E}\overline{\langle A\rangle_{\mathbf{x}}} \;\Rightarrow\; \overline{\mathbb{E}\langle A\rangle_{\mathbf{x}}}\,)$$

If $\mathbb{E}$ were not a weird operator (see Section 5.4.4.3), $\mathbb{E}\overline{\langle A\rangle_{\mathbf{x}}}$ would be equivalent to $\overline{\mathbb{E}\langle A\rangle_{\mathbf{x}}}$; and we expect that equivalence to be true for most actions $\langle A\rangle_{\mathbf{x}}$. However, because it is not always true, we have to add (7.15) as an assumption.

To see what is required to make (7.14b) true, we consider what assumption is required to ensure that $\overline{P} \Rightarrow P$ is true for an arbitrary state predicate P with free variables $\mathbf{x}$. The free variables of $\overline{P}$ are $\mathbf{X}$, and the relation between the values of $\mathbf{x}$ and $\mathbf{X}$ is described by the invariant I^R of $S \otimes S^R$.

There are three cases, depending on whether $\mathcal{R}$ is true, $\mathcal{L}$ is true, or neither is true:

$\mathcal{R}$ true: The conjunct

$$\mathcal{R} \Rightarrow ([R]^+_{\mathbf{x}} \text{ AWITH } \mathbf{x} \leftarrow \mathbf{X}, \mathbf{x}' \leftarrow \mathbf{x})$$

of I^R asserts that, in this case, we can arrive at the values of $\mathbf{x}$ in the current state by starting in a state in which the values of $\mathbf{x}$ equal the current values of $\mathbf{X}$ and executing a sequence of R steps. This means that $\overline{P} \Rightarrow P$ is true if, starting in a state satisfying P and executing a sequence of R steps, we reach a state in which P is true. This is true iff, starting in a state satisfying P and repeatedly executing single R steps, we keep reaching states satisfying P. In other words, if $P \wedge R \Rightarrow P'$ is true in any state satisfying $\mathcal{R}$ of a behavior of $S \otimes S^R$, then $\overline{P} \Rightarrow P$ is true.

$\mathcal{L}$ true: A similar argument based on the conjunct

$$\mathcal{L} \Rightarrow ([L]^+_{\mathbf{x}} \text{ AWITH } \mathbf{x}' \leftarrow \mathbf{X})$$

of I^R shows that if $L \wedge P' \Rightarrow P$ is true in any state satisfying $\mathcal{L}$ in a behavior of $S \otimes S^R$, then $\overline{P} \Rightarrow P$ is true.

Neither $\mathcal{R}$ nor $\mathcal{L}$ true: The conjunct $\neg(\mathcal{R} \vee \mathcal{L}) \Rightarrow (\mathbf{X} = \mathbf{x})$ of I^R shows that $\overline{P} \equiv P$ is true for any state satisfying $\neg(\mathcal{R} \vee \mathcal{L})$ in a behavior of $S \otimes S^R$

Let's review what we have shown. We can deduce (7.10) from (7.11) and (7.12). If (7.13) is true, then we can choose S^R of Theorem 7.1 to make (7.11) true for a single subaction A of E. We can deduce (7.12) from (7.14a) and (7.14b). We can deduce (7.14a) from (7.15). And finally, we can deduce (7.14b) from the conditions obtained above for proving $\overline{P} \Rightarrow P$, substituting $\mathbb{E}\langle A \rangle_{\mathbf{x}}$ for P. Putting all this together, we have shown that the program S^R of Theorem 7.1 can be chosen to make (7.10) true, for a single subaction A of E, if the following two conditions are satisfied:

$$(7.16)\quad \begin{aligned} &\models S \;\Rightarrow\; \Box\,[\,\wedge\,(R \cdot \langle A \rangle_{\mathbf{x}} \Rightarrow \langle A \rangle_{\mathbf{x}} \cdot R) \,\wedge\, (\langle A \rangle_{\mathbf{x}} \cdot L \Rightarrow L \cdot \langle A \rangle_{\mathbf{x}}) \\ &\qquad\qquad\quad \wedge\, \mathbb{E}\langle A \rangle_{\mathbf{x}} \wedge R \;\Rightarrow\; (\mathbb{E}\langle A \rangle_{\mathbf{x}})' \\ &\qquad\qquad\quad \wedge\, L \wedge (\mathbb{E}\langle A \rangle_{\mathbf{x}})' \;\Rightarrow\; \mathbb{E}\langle A \rangle_{\mathbf{x}}\;]_{\mathbf{x}} \\ &\models S^R \;\Rightarrow\; \Box\,(\overline{\mathbb{E}\langle A \rangle_{\mathbf{x}}} \;\Rightarrow\; \overline{\mathbb{E}\langle A \rangle_{\mathbf{x}}}) \end{aligned}$$

7.1.5.2 Fairness of Subactions of M^R

Deducing fairness properties of subactions of M^R from fairness properties of actions of S is more complicated than for subactions of E^R because an M^R step is the result of executing multiple actions of S as a single step. The simplest and probably most common case is when we want the reduced program to satisfy fairness of the M^R action itself. So, we begin by examining this case.

Action M^R is the action obtained by executing the entire operation consisting of a sequence of R steps, followed by a C step, followed by a complete sequence of L steps, with the variables $\mathbf{X}$ substituted for the variables $\mathbf{x}$. For our example program of Figure 7.1, where the values of variables $\mathbf{X}$ in state s were defined to be the values of $\mathbf{x}$ in state $\phi(s)$, condition $\phi3$ implies that M^R equals $\overline{C}$. This means that in any behavior of $S \otimes S^R$, a C step is an M^R step. It suggests that we should be able to deduce fairness of the M^R action of S^R from fairness of the C action of S.

Now suppose A is a subaction of C and A^R is the action obtained by executing the entire operation consisting of a sequence of R steps, followed by an A step, followed by a complete sequence of L steps, with the variables $\mathbf{X}$ substituted for the variables $\mathbf{x}$. The same argument shows that A^R equals $\overline{A}$, and that in any behavior of $S \otimes S^R$, an A step is an A^R step. Moreover, any A^R step is also an M^R step, so A^R is a subaction of M^R. So, it is reasonable to consider deriving fairness of a subaction A^R of M^R when A^R is obtained in this way from a subaction A of C. Since our goal is not to obtain the most general results, we consider only this case.

First, we must define A^R precisely for a subaction A of C. An A^R step is obtained by combining a sequence of R steps followed by an A step followed by a sequence of L steps into a single step, and then substituting $\mathbf{X}$ for $\mathbf{x}$. The definitions of R, C, and L in terms of $\mathcal{R}$ and $\mathcal{L}$ and assumptions M1–M4 imply that A^R equals $\overline{A^\rho}$ where A^ρ is defined by

$$(7.17)\quad A^\rho \;\triangleq\; \neg(\mathcal{R} \lor \mathcal{L}) \;\land\; ([R]^+_{\mathbf{x}} \cdot A \cdot [L]^+_{\mathbf{x}}) \;\land\; \neg(\mathcal{R} \lor \mathcal{L})'$$

We want condition (7.10) to be true for this choice of A and A^R. As before, we do this by making (7.11) and (7.12) true, starting with (7.11). In a behavior satisfying $S \otimes S^R$, every A step is an A^R step. However, (7.11) requires every $\langle A \rangle_{\mathbf{x}}$ step to be an $\langle A^R \rangle_{\mathbf{X}}$ step. This is not true for A an arbitrary subaction of C. It's not even necessarily true for $A = C$, for the following reason.

Recall that Figure 7.4 shows two behaviors that satisfy S, the bottom one being the reduced version of the top one. The action labels describe the changes of the values of the variables $\mathbf{x}$. The top behavior satisfies $S \otimes S^R$, where the values of variables $\mathbf{X}$ in a state s_i of that behavior are the values of $\mathbf{x}$ in the corresponding state $\phi(s_i)$ of the bottom sequence. The step $s_{43} \rightarrow s_{44}$ of the top behavior is a $\overline{C}$ step iff $u_{42} \rightarrow u_{46}$ is a C step. If one of the variables $\mathbf{x}$ has different values in states s_{43} and s_{44}, there is no reason why its value should differ in states u_{42} and u_{46}. Step $s_{43} \rightarrow s_{44}$ would be a $\langle C \rangle_{\mathbf{x}}$ step but not a $\langle \overline{C} \rangle_{\mathbf{X}}$ step if the values of all the variables $\mathbf{x}$ are the same in states u_{42} and u_{46}. To be able to deduce (7.11) when A is C and A^R is M^R, we need the assumption that if $s_{43} \rightarrow s_{44}$ is a $\langle C \rangle_{\mathbf{x}}$ step, then $\mathbf{x}' \neq \mathbf{x}$ is true of step $u_{42} \rightarrow u_{46}$.

Since every A step in a behavior of $S \otimes S^R$ is an A^R step, we can deduce that every $\langle A \rangle_{\mathbf{x}}$ step is an $\langle A^R \rangle_{\mathbf{X}}$ step from this assumption:

$$\models S \;\Rightarrow\; \Box[\,(\langle A \rangle_{\mathbf{x}})^{\rho} \Rightarrow (\mathbf{x}' \neq \mathbf{x})\,]_{\mathbf{x}}$$

There is seldom any reason for a program's next-state action to allow stuttering steps, and modifying it to disallow stuttering steps does not change the program. An A step of the program will usually be an $\langle A \rangle_{\mathbf{x}}$ step; and if it isn't, A can be replaced by $\langle A \rangle_{\mathbf{x}}$. So for simplicity, we strengthen this assumption to:

$$(7.18) \quad \models S \;\Rightarrow\; \Box[A^{\rho} \Rightarrow (\mathbf{x}' \neq \mathbf{x})]_{\mathbf{x}}$$

We have shown that (7.18) implies that in a behavior of $S \otimes S^R$, every $\langle A \rangle_{\mathbf{x}}$ step is an $\langle A^R \rangle_{\mathbf{X}}$ step. In other words, we have shown that it implies:

$$(7.19) \quad \models S \otimes S^R \;\Rightarrow\; \Box[\,\langle A \rangle_{\mathbf{x}} \Rightarrow \langle A^R \rangle_{\mathbf{X}}\,]_{\mathbf{x},\mathbf{X}}$$

This assertion implies (7.11).

The assumption (7.18) makes (7.11) true, so we now have to make (7.12) true. But we can't expect (7.12) to hold in general for the following reason. Since A^R equals $\overline{A^{\rho}}$, we expect $\mathbb{E}\langle A^R \rangle_{\mathbf{X}}$ to imply $\mathbb{E}\langle A^{\rho} \rangle_{\mathbf{x}}$. By (7.17), $\mathbb{E}\langle A^{\rho} \rangle_{\mathbf{x}}$ equals $\mathbb{E}\langle \neg(\mathcal{R} \vee \mathcal{L}) \wedge [R]^{+}_{\mathbf{x}} \cdot A \cdot \ldots \rangle_{\mathbf{x}}$, which implies $\mathbb{E}(\neg(\mathcal{R} \vee \mathcal{L}) \wedge (R \vee A))$ (since $\mathbb{E}(U \cdot V)$ implies $\mathbb{E}(U)$ for any actions U and V). We can therefore expect $\mathbb{E}\langle A^R \rangle_{\mathbf{X}}$ to imply $\mathbb{E}\langle A \rangle_{\mathbf{x}}$, as required by (7.12), only when there is no R action.

A sequence of R steps may have to occur between when A^{ρ} becomes enabled and when A becomes enabled, so fairness assumptions for R as well as a fairness assumption for A may be required to imply fairness of A^{ρ}. Instead of assuming a fairness condition on L actions to ensure that operation M completes after a C step occurs, we simply assumed that the operation completes by adding the assumption $\Box\Diamond\neg\mathcal{L}$. Similarly, instead of assuming a fairness condition on R actions to ensure that the necessary enabling condition of C occurs, we simply assume that the enabling condition eventually occurs.

As an example, suppose we want to deduce weak fairness of A^R from strong fairness of A. (Because fairness of A^R requires fairness of more than just A, there's no reason not to have different kinds of fairness for the two actions.) By (7.19) and the definition (4.12) of WF, to prove $\mathrm{WF}_{\mathbf{X}}(A^R)$ it suffices to prove $\Diamond\Box\,\mathbb{E}\langle A^R\rangle_{\mathbf{X}} \leadsto \langle A\rangle_{\mathbf{x}}$. Just as we split the proof of (7.12) into the two conditions of (7.14), we split the proof of $\Diamond\Box\,\mathbb{E}\langle A^R\rangle_{\mathbf{X}} \leadsto \langle A\rangle_{\mathbf{x}}$ into proving:

(7.20) (a) $\Box(\mathbb{E}\langle A^R\rangle_{\mathbf{X}} \Rightarrow \overline{\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}}}\,)$
(b) $\Diamond\Box\,\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}} \leadsto \langle A\rangle_{\mathbf{x}}$

This may seem wrong because we have $\overline{\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}}}$ in (7.20a) and $\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}}$ in (7.20b) when the two formulas should be equal. However, the following reasoning shows that they are equal. The definition of A^ρ and conditions $\mathbb{E}3$ and $\mathbb{E}4$ of Section 5.4.4.2 imply that $\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}}$ equals $\neg(\mathcal{R}\vee\mathcal{L})\wedge\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}}$. The invariant I^R implies that $\neg(\mathcal{R}\vee\mathcal{L}) \Rightarrow (\mathbf{x} = \mathbf{X})$ and $\overline{\neg(\mathcal{R}\vee\mathcal{L})}$ are true, so $S\otimes S^R$ implies that $\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}}$ always equals $\overline{\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}}}$. We make S^R implying (7.20a) one of our requirements for deducing that $\mathrm{WF}_{\mathbf{X}}(A^R)$ is satisfied. We now consider (7.20b).

By (3.32b) of Section 3.4.2.8 and the tautology $\models \neg\langle A\rangle_{\mathbf{x}} \equiv [\neg A]_{\mathbf{x}}$, to prove (7.20b) it suffices to prove:

(7.21) $\Diamond\Box\,\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}} \wedge \Box[\neg A]_{\mathbf{x}} \leadsto \langle A\rangle_{\mathbf{x}}$

By the definition (4.21) of SF and transitivity of $\leadsto$, to deduce (7.21) from $\mathrm{SF}_{\mathbf{x}}(A)$, it suffices to prove that S implies:

(7.22) $\Diamond\Box\,\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}} \wedge \Box[\neg A]_{\mathbf{x}} \leadsto \Box\Diamond\,\mathbb{E}\langle A\rangle_{\mathbf{x}}$

We have seen that to deduce $\mathrm{WF}_{\mathbf{X}}(A^R)$ from $\mathrm{SF}_{\mathbf{x}}(A)$, it suffices to show (7.18) and:

(7.23) (a) $\models S^R \;\Rightarrow\; \Box(\mathbb{E}\langle A^R\rangle_{\mathbf{X}} \Rightarrow \overline{\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}}})$
(b) $\models S \;\Rightarrow\; (\Diamond\Box\,\mathbb{E}\langle A^\rho\rangle_{\mathbf{x}} \wedge \Box[\neg A]_{\mathbf{x}} \leadsto \Box\Diamond\,\mathbb{E}\langle A\rangle_{\mathbf{x}})$

For the other three possible pairs of fairness conditions on A^R and A, the same argument shows that we can deduce $\mathrm{SF}_{\mathbf{X}}(A^R)$ instead of $\mathrm{WF}_{\mathbf{X}}(A^R)$ by replacing $\Diamond\Box$ with $\Box\Diamond$ in (7.23b); and we can assume $\mathrm{WF}_{\mathbf{x}}(A)$ instead of $\mathrm{SF}_{\mathbf{x}}(A)$ by replacing $\Box\Diamond$ with $\Diamond\Box$ in (7.23b).

7.1.5.3 The Reduction Theorem with Fairness

We have described the assumption needed to infer that the reduced program satisfies a fairness condition on a single action A^R if the original program satisfies a fairness condition on a single action A. We now combine this into a theorem for inferring that the reduced program satisfies the conjunction of countably many fairness conditions A_i^R. This is simple, except for one problem mentioned above for the case in which A^R is a subaction of E^R.

Recall that to satisfy (7.11), we constrained the construction of the function ϕ illustrated in Figures 7.2–7.4 so that if an E step is an A step in the original behavior, then the corresponding E step in the reduced behavior is also an A step. With multiple actions A_i, if the E step in the original behavior is both an A_i and an A_j step for $j \neq i$, it might be impossible to make the E step in the transformed behavior both an A_i and an A_j step.

However, to satisfy (7.11), it's not necessary for every E step that's an A step in the original behavior to remain an A step in the reduced behavior. It's only necessary to ensure that if there are infinitely many A steps in the original behavior, then infinitely many of them are A steps in the reduced behavior. With countably many such actions A_i, it's possible to construct the reduced behavior so that for every i for which there are infinitely many A_i steps in the original behavior, there are infinitely many A_i steps in the reduced behavior. This is done using Lemma B.1 in Appendix Section B.6 the same way it is used there in the proof of Theorem 4.6.

We can now put together the assumptions we derived above for deducing fairness of an action A^R from fairness of an action A for individual actions A into a theorem for deducing fairness of a countable number of actions A_i^R from fairness of actions A_i. The requirements for A_i a subaction of E (and A_i^R a subaction of E^R) come from (7.16). The requirements for A_i a subaction of C (and A_i^R a subaction of M^R) come from (7.18) and (7.23) plus the modification of (7.23) for additional fairness conditions of A and A^R.

Theorem 7.4 With the definitions and assumptions (1) and (2) of Theorem 7.1, let $C \triangleq (\neg\mathcal{L}) \land M \land (\neg\mathcal{R}')$ and let

$$\models F \;\Rightarrow\; \forall\, i \in I \,:\, \mathrm{YF}^i_{\mathbf{x}}(A_i) \qquad F^R \;\triangleq\; \forall\, i \in I \,:\, \mathrm{ZF}^i_{\mathbf{X}}(A_i^R)$$

where I is a countable set and YF^i and ZF^i are WF or SF for each $i \in I$; and assume either:

- A_i is a subaction of E, $A_i^R \triangleq \overline{A_i}$, YF^i equals ZF^i,

$$\begin{aligned}\models S \;\Rightarrow\; \Box\,[\,&\land\;(R \cdot \langle A_i\rangle_{\mathbf{x}} \Rightarrow \langle A_i\rangle_{\mathbf{x}} \cdot R) \;\land\; (\langle A_i\rangle_{\mathbf{x}} \cdot L \Rightarrow L \cdot \langle A_i\rangle_{\mathbf{x}})\\ &\land\; \mathbb{E}\langle A_i\rangle_{\mathbf{x}} \land R \;\Rightarrow\; (\mathbb{E}\langle A_i\rangle_{\mathbf{x}})'\\ &\land\; L \land (\mathbb{E}\langle A_i\rangle_{\mathbf{x}})' \;\Rightarrow\; \mathbb{E}\langle A_i\rangle_{\mathbf{x}}\,]_{\mathbf{x}}\,, \quad \text{and}\end{aligned}$$

$$\models S^R \;\Rightarrow\; \Box\,(\mathbb{E}\overline{\langle A_i\rangle_{\mathbf{x}}} \;\Rightarrow\; \overline{\mathbb{E}\langle A_i\rangle_{\mathbf{x}}})$$

or

- A_i is a subaction of C,

  $A_i^{\rho} \;\triangleq\; \neg(\mathcal{R} \lor \mathcal{L}) \;\land\; ([R]^{+}_{\mathbf{x}} \cdot A_i \cdot [L]^{+}_{\mathbf{x}}) \;\land\; \neg(\mathcal{R} \lor \mathcal{L})'$,

  $A_i^{R} \;\triangleq\; \overline{A_i^{\rho}}\,$,

  $\models S \;\Rightarrow\; \Box[A_i^{\rho} \Rightarrow (\mathbf{x}' \neq \mathbf{x})]_{\mathbf{x}}\,$,

  $\models S^R \;\Rightarrow\; \Box(\mathbb{E}\langle A_i^{R}\rangle_{\mathbf{X}} \Rightarrow \overline{\mathbb{E}\langle A_i^{\rho}\rangle_{\mathbf{x}}})\,$, and

  $\models S \land F \;\Rightarrow\; (\boxed{\diamond}\boxed{\diamond}^{\,\mathrm{Z}}\,\mathbb{E}\langle A_i^{\rho}\rangle_{\mathbf{x}} \land \Box[\neg A_i]_{\mathbf{x}} \leadsto \boxed{\diamond}\boxed{\diamond}^{\,\mathrm{Y}}\,\mathbb{E}\langle A_i\rangle_{\mathbf{x}})$

  where for Q either Y or Z, $\boxed{\diamond}\boxed{\diamond}^{\,\mathrm{Q}}$ is $\Diamond\Box$ if QF is WF, and it is $\Box\Diamond$ if QF is SF.[3]

Then $\models S \land F \land \Box\Diamond\neg\mathcal{L} \;\Rightarrow\; \boldsymbol{\exists}\,\mathbf{X} \,:\, S^R \land \Box I^R \land F^R$.

7.1.6 An Example: Making Critical Sections Atomic

A standard concurrent coding practice is to "protect" accesses to shared data by putting them in critical sections. Recall that Section 4.2.2.1 defined a critical section to be a piece of code in a process such that no two processes can be executing their critical sections at the same time. We can consider this coding practice to be an application of reduction in which the reduced program executes the entire critical section, including its waiting and exiting code, as a single action.

We assume that critical sections are implemented with the trivial mutual exclusion algorithm *LM* described in Figure 4.6 of Section 4.2.6.1 that uses Dijkstra's P and V semaphore operations. (Correctness of a mutual exclusion algorithm can be expressed as the requirement that it implements *LM* under a suitable refinement mapping.) We assume the program is described with pseudocode, using a semaphore variable *sem* and a variable *pc* to describe the control state. The variable *sem* initially equals 0 and is accessed by a process p only by P_p and V_p actions that execute the $P(\mathit{sem})$ and $V(\mathit{sem})$ operations. These actions are written in TLA as follows, where the

[3]For example, if $\mathrm{YF}^{i}_{\mathbf{x}}(A_i)$ is $\mathrm{SF}^{i}_{\mathbf{x}}(A_i)$ and $\mathrm{ZF}^{i}_{\mathbf{X}}(A_i^R)$ is $\mathrm{WF}^{i}_{\mathbf{X}}(A_i^R)$, this condition is:

$$\models S \land F \;\Rightarrow\; (\Diamond\Box\,\mathbb{E}\langle A_i^{\rho}\rangle_{\mathbf{x}} \land \Box[\neg A_i]_{\mathbf{x}} \leadsto \Box\Diamond\,\mathbb{E}\langle A_i\rangle_{\mathbf{x}})$$

UNCHANGED formulas assert that all program variables other than *sem* and *pc* are left unchanged:

$$\begin{array}{lcl} P_p & \triangleq & \land\ pc(p) = \ldots \\ & & \land\ (sem = 1) \land (sem' = 0) \\ & & \land\ pc' = (pc \text{ EXCEPT } p \mapsto \ldots) \\ & & \land\ \text{UNCHANGED } \ldots \\ V_p & \triangleq & \land\ pc(p) = \ldots \\ & & \land\ sem' = 1 \\ & & \land\ pc' = (pc \text{ EXCEPT } p \mapsto \ldots) \\ & & \land\ \text{UNCHANGED } \ldots \end{array}$$

The value of *sem* is therefore always either 0 or 1. An execution of a critical section by process p consists of a P_p step, followed by steps satisfying actions $CS_{p,1}$, ..., CS_{p,k_p} that represent executions of statements in the critical section, followed by a V_p step. The assumption that shared data is accessed only within a critical section means that if an action E_q describes a statement of process q outside a critical section (meaning not a P_q, $CS_{q,j}$, or V_q action), then each action $CS_{p,i}$ commutes with E_q if $p \neq q$.

We can reduce a program using critical sections in this way by making execution of each critical section a single step. This is done by a sequence of applications of our reduction theorems, each one reducing one critical section of a single process. When doing multiple reductions, it would get confusing if we introduced new variables for each reduction. We therefore consider the reduced version S^R of a program S to be the program S^R described in our theorems, except with $\mathbf{x}$ substituted for $\mathbf{X}$. In our theorems, the M^R that executes the operation described by action M as a single step is defined to equal M^ρ WITH $\mathbf{x} \leftarrow \mathbf{X}$. When we use the variables $\mathbf{x}$ instead of $\mathbf{X}$ for the reduced program, M^R becomes M^ρ.

To apply the theorems to a critical section of a process p, we let M_p equal $P_p \lor CS_{p,1} \lor \ldots \lor CS_{p,k_p} \lor V_p$, so M^p is replaced in the reduced program by M_p^ρ. We could let any of those actions be the action C, but it is most convenient to let C be the P_p action, so there is no action R and action L is $CS_{p,1} \lor \ldots \lor CS_{p,k_p} \lor V_p$. By Theorem 7.2, it suffices to show that V_p and each $CS_{p,i}$ left-commutes with every action of every process $q \neq p$. (Action L trivially left-commutes with any action of process p not a subaction of M_p.) The possible actions of process q are P_q, V_q, $CS_{q,j}$ for some j, an action E_q not in a critical section, and M_q^ρ if the operation described by M_q has already been reduced. Here is why V_p and $CS_{p,i}$ commute with each of those actions of process $q \neq p$:

P_q: V_p left-commutes with P_q because the mutual exclusion algorithm implies that V_p is not enabled when a process $q \neq p$ is in its critical section, so $P_q \cdot V_p$ equals FALSE; $CS_{p,i}$ commutes with P_q because it does not access *sem* or $pc(q)$ (meaning that it does not depend on or modify *sem* or $pc(q)$).

V_q: V_p commutes with V_q because any two $V(sem)$ operations commute; $CS_{p,i}$ commutes with V_p because it does not access *sem* or $pc(q)$.

$CS_{q,j}$: $CS_{q,j} \cdot V_p$ and $CS_{q,j} \cdot CS_{p,i}$ both equal FALSE because a $CS_{q,j}$ step leaves process q inside its critical section, which by the mutual exclusion algorithm implies process q is outside its critical section so neither $CS_{p,i}$ nor V_p is enabled.

E_q: V_p commutes with E_q because E_q does not access *sem* or $pc(p)$; and $CS_{p,i}$ commutes with E_q because of the assumption that actions describing a statement inside a critical section commute with all actions describing statements not in another process's critical section.

M_q^ρ: Both $M_q^\rho \cdot V_p$ and $M_q^\rho \cdot CS_{p,i}$ equal FALSE because an M_q^ρ step leaves $sem = 1$, while the mutual exclusion algorithm implies that V_p and $CS_{p,i}$ are enabled only when $sem = 0$.

This shows that S implies $E \cdot L \Rightarrow L \cdot E$, so since there is no R action, we can apply Theorem 7.1. There is still the formula $\square\diamond\neg\mathcal{L}$ to deal with. If the program has fairness assumptions, then we expect $\square\diamond\neg\mathcal{L}$ to be implied by fairness assumptions of the L actions. If not, then by Theorem 7.3 we can verify safety properties by showing that, after executing process p's P_p statement, it is always possible for execution of the critical section to complete. It's hard to imagine an application of mutual exclusion that would allow a behavior in which it is impossible for some process ever to exit its critical section.

We now consider deducing fairness properties of the reduced program. First, let action A describe a statement outside any critical section. If the statement is in process p, then A^ρ equals A so fairness of A^ρ is equivalent to fairness of A. If A is an action of a process other than p, the argument above shows that L commutes with $\langle A\rangle_{\mathbf{x}}$. If $\langle A\rangle_{\mathbf{x}}$ is a subaction of the next-state action of S, then an $L \wedge (\mathbb{E}\langle A\rangle_{\mathbf{x}})'$ step ends in a state in which an $\langle A\rangle_{\mathbf{x}}$ step can occur, which by commutativity of L and $\langle A\rangle_{\mathbf{x}}$ implies it begins in a state in which an $\langle A\rangle_{\mathbf{x}}$ step can occur, so S implies $\square(L \wedge (\mathbb{E}\langle A\rangle_{\mathbf{x}})' \Rightarrow \mathbb{E}\langle A\rangle_{\mathbf{x}})$. We expect $\square\,(\mathbb{E}\overline{\langle A_i\rangle_{\mathbf{x}}} \;\Rightarrow\; \overline{\mathbb{E}\langle A_i\rangle_{\mathbf{x}}})$ to be true, since substitution has been found to distribute over $\mathbb{E}$ for most actions. (However, there is little experience with the substitution $\mathbf{x} \leftarrow \mathbf{X}$ that occurs in reduction.) If this formula is true, then by Theorem 7.4 we can deduce fairness of A^ρ from fairness of A.

We expect the most common fairness property for a subaction of M_p^ρ to be fairness of M_p^ρ itself. In this case C is P_p, and we expect to deduce a fairness condition of M_p^ρ from the same fairness condition of the P_p action. There is no R action, so the definitions of P_p and M_p^ρ imply $M_p^\rho \Rightarrow (\mathbf{x}' \neq \mathbf{x})$ and $\mathbb{E}\langle M_p^\rho\rangle_{\mathbf{x}} \Rightarrow \mathbb{E}\langle P_p\rangle_{\mathbf{x}}$. (In the unlikely case that process p does nothing but repeatedly execute the critical section, so M_p^ρ leaves $pc(p)$ unchanged, we need to add the assumption that M_p^ρ changes some other variable.) The only remaining requirement to deduce from Theorem 7.4 that fairness of P_p implies fairness of M_p^ρ is:

$$\models S^R \;\Rightarrow\; \Box(\mathbb{E}\langle M^R\rangle_{\mathbf{X}} \Rightarrow \overline{\mathbb{E}\langle M^\rho\rangle_{\mathbf{x}}})$$

Since $\langle M^R\rangle_{\mathbf{x}}$ equals $\overline{\langle M^\rho\rangle_{\mathbf{x}}}$, this is true if substitution distributes over $\mathbb{E}$ for the action $\langle M^\rho\rangle_{\mathbf{x}}$.

We have described the mathematics underlying the use of mutual exclusion to implement atomic operations. We have ignored the question of what that achieves. We use mutual exclusion to view execution of the critical section as a single step, but is that a correct view? The answer lies in condition R3 of Section 7.1.1.2, which tells us when we can deduce that the original program satisfies a property that the reduced program satisfies. The only part of the invariant I^R that seems useful in condition R3 is the conjunct $\neg(\mathcal{R} \vee \mathcal{L}) \Rightarrow (\mathbf{X} = \mathbf{x})$, which asserts that the variables of the original program and the reduced one have equal values when no process is executing its critical section.

7.1.7 Another Example: Pipelining

Here is a sketch of a simple example of reduction that is interesting in part because each operation being reduced contains actions performed by two different processes. The example is a very abstract view of one stage in a pipelined computation—a view that tells us nothing about what is being computed.

The program S performs a sequence of computations. Each computation presumably obtains some input, does some computation, and then produces some output—but that is irrelevant. What concerns us is that the computations are pipelined as follows so they can be performed concurrently by two processes. Process 1 performs the first part of the computation to obtain a partial result that it appends to the end of a FIFO queue. Process 2 removes the partial result from the head of the queue and completes the computation. Process 1 can therefore get ahead of process 2, performing its part of the i^{th} computation while process 2 is still performing its part of the j^{th} computation, for $i > j$. The reduced program S^R replaces these two processes with a single process that performs each computation as a single atomic action. The property we want to prove by reduction presumably involves how the computed values are used after they are computed, when they have the same values in the original and reduced programs, so condition R3 is satisfied.

We describe steps of process 1 by an action $Cmp1 \lor Send$, where that process's part of a computation consists of a finite sequence of $Cmp1$ steps followed by a single $Send$ step that appends the partial result to the tail of the queue. Steps of process 2 are described by an action $Rcv \lor Cmp2$, where that process's part of the computation consists of a single Rcv step that removes the partial result from the head of the queue followed by a finite sequence of $Cmp2$ steps. The contents of the queue are described by the variable $qBar$, which is accessed only by the $Send$ and Rcv actions. We assume that the two processes communicate only through the FIFO $qBar$, an assumption expressed by these conditions: $Cmp1$ commutes with the process 2 actions Rcv and $Cmp2$, and $Cmp2$ commutes with the process 1 actions $Cmp1$ and $Send$. Since $qBar$ is the only shared variable accessed by Rcv and $Send$, it doesn't matter in which order these two actions are executed in a state where the queue is nonempty. Thus, we have:

$$(7.24) \quad \models (qBar \neq \langle\,\rangle) \Rightarrow (Send \cdot Rcv \equiv Rcv \cdot Send)$$

The program may contain other processes that can interact in some way with processes 1 and 2. For example, process 1 may obtain its input from a third process and process 2 may send its output to a fourth process.

The program's next-state action is $M \lor O$, where M describes processes 1 and 2 and O describes any other processes. We rewrite M in the form $\exists\, n \in \mathbb{N}^+ : M_n$, where $\mathbb{N}^+$ is the set of positive integers and M_n is an action whose steps describe a complete execution of the n^{th} computation. To do this, we assume state functions $snum$ and $rnum$ whose values are the numbers of $Send$ and Rcv actions, respectively, that have been executed. Initially, $snum = rnum = 0$. The $Send$ action increments $snum$ by 1 and the Rcv action increments $rnum$ by 1. We can then define:

$$\begin{aligned}(7.25)\quad M_n \;\triangleq\;& \lor\ (snum = n-1) \land (Cmp1 \lor Send)\\ & \lor\ ((rnum = n-1) \land Rcv) \lor ((rnum = n) \land Cmp2)\end{aligned}$$

We recursively define the n^{th} reduction of the program to be the one obtained by reducing the operations $M_1, \ldots, M_n$ in that order. To define the n^{th} reduction, define:

$$\begin{aligned}
\mathit{Cmp}1_n &\triangleq (\mathit{snum} = n-1) \land \mathit{Cmp}1 \\
\mathit{Send}_n &\triangleq (\mathit{snum} = n-1) \land \mathit{Send} \\
\mathit{Rcv}_n &\triangleq (\mathit{rnum} = n-1) \land \mathit{Rcv} \\
\mathit{Cmp}2_n &\triangleq (\mathit{rnum} = n) \land \mathit{Cmp}2
\end{aligned}$$

so M_n equals $\mathit{Cmp}1_n \lor \mathit{Send}_n \lor \mathit{Rcv}_n \lor \mathit{Cmp}2_n$. The actions R, C, and L for the n^{th} reduction are:

$$R_n \triangleq \mathit{Cmp}1_n \qquad C_n \triangleq \mathit{Send}_n \qquad L_n \triangleq \mathit{Rcv}_n \lor \mathit{Cmp}2_n$$

Again, with multiple reductions we let the reduced program have the same variables as the original program, so the n^{th} reduction replaces the action M_n with M_n^{ρ}.

The remaining action E for this reduction is the disjunction of these actions: the action O describing the other processes, the already reduced actions M_k^{ρ} for $k < n$, and the subactions of M_k for $k > n$. To apply Theorem 7.1, we must show that R_n right-commutes with these actions and L_n left-commutes with them.

That R_n right-commutes and L_n left-commutes with O must be assumed. The commutativity relations hold for M_k^{ρ} with $k < n$ because an R_n step is enabled only after an M_k^{ρ} step, which implies $R_n \cdot M_k^{\rho}$ equals FALSE (so R_n right commutes with M_k^{ρ}), and which also implies that L_n cannot be enabled immediately after an M_k^{ρ} step, so $M_k^{\rho} \cdot L_n$ also equals FALSE.

What remains to be shown is that $\mathit{Cmp}1_n$ (the action R_n) right commutes with M_k, and that Rcv_n and $\mathit{Cmp}2_n$ (whose disjunction equals L_n) left commutes with M_k, for $k > n$. For that, we have to show that each of the four actions whose disjunction equals M_k satisfy those commutativity conditions. We will use the commutativity relations we assumed above: that $\mathit{Cmp}1$ commutes with $\mathit{Cmp}2$ and Rcv, and that $\mathit{Cmp}2$ commutes with Send. The assumption that $\mathit{Cmp}2$ commutes with Send implies that $\mathit{Cmp}2_i$ commutes with Send_j for all i and j. This follows from the definitions of $\mathit{Cmp}2_i$ and Send_j, because $\mathit{Cmp}2$ does not depend on or modify snum and Send_j does not depend on or modify rnum. Similarly, $\mathit{Cmp}1_i$ commutes with Rcv_j and $\mathit{Cmp}2_j$ for all i and j. These assumptions are called the *commutativity assumptions* in the following proof sketches of the required commutativity relations. Recall that we are assuming $k > n$.

$Cmp1_n$ **right-commutes with** $Cmp1_k$ **and** $Send_k$

$Cmp1_n \cdot Cmp1_k$ and $Cmp1_n \cdot Send_k$ equal FALSE because $snum = n-1$ after a $Cmp1_n$ step; $Cmp1_k$ and $Send_k$ are enabled iff $snum = k-1$; and $k > n$.

$Cmp1_n$ **right-commutes with** Rcv_k **and** $Cmp2_k$

By the commutativity assumptions.

Rcv_n **left-commutes with** $Cmp1_k$

By the commutativity assumptions.

Rcv_n **left-commutes with** $Send_k$

A $Send_k$ step is enabled only if $snum = k-1$; the step leaves $rnum$ unchanged; and Rcv_n is enabled only if $rnum = n-1$. Therefore, $Send_k \cdot Rcv_n$ enabled implies $snum = k-1$ and $rnum = n-1$, so $k > n$ implies $snum > rnum$ which implies $qBar \neq \langle\,\rangle$. By (7.24), this implies $Send_k \cdot Rcv_n \equiv Rcv_n \cdot Send_k$.

Rcv_n **left-commutes with** Rcv_k **and** $Cmp2_k$

$Rcv_k \cdot Rcv_n$ and $Cmp2_k \cdot Rcv_n$ equal FALSE because a Rcv_k or $Cmp2_k$ step ends in a state with $rnum = k$ which by $k > n$ implies $rnum \neq n-1$, so Rcv_n is not enabled in that state.

$Cmp2_n$ **left-commutes with** $Cmp1_k$ **and** $Send_k$

By the commutativity assumptions.

$Cmp2_n$ **left-commutes with** Rcv_k **and** $Cmp2_k$

A Rcv_k or $Cmp2_k$ step ends with $rnum = k$, and $Cmp2_n$ is enabled only if $rnum = n$, which is false because $k > n$. Therefore a $Cmp2_n$ step cannot follow a Rcv_k or $Cmp2_k$ step, so $Rcv_k \cdot Cmp2_n$ and $Cmp2_k \cdot Cmp2_n$ equal FALSE.

This handles the commutativity assumptions of Theorem 7.1. We still have the hypothesis $\Box\Diamond\neg\mathcal{L}$ in the theorem's conclusion to deal with. By Theorem 7.3, to use reduction for verifying safety properties, we don't need that hypothesis. We only have to show that any finite behavior satisfying the safety property S can be extended to a behavior that completes each M_n operation in which the C_n action has occurred. This means showing that Process 2 cannot deadlock. This should be easy to show unless process 2 may have to wait for another process to do something—for example, until another process is ready to receive process 2's output.

For reasoning about liveness, we would expect the hypothesis $\Box\Diamond\neg\mathcal{L}$ to be satisfied by adding fairness conditions to process 2 actions, and perhaps to other processes, to ensure that the operation will complete once process 1's *Send* action occurs. The obvious fairness condition we want the reduced program to satisfy is fairness of M^ρ. If an M^ρ_n action is enabled, then no M^ρ_i action with $i \neq n$ can be enabled until an M^ρ_n step occurs. This implies that (weak or strong) fairness of M^ρ is equivalent to fairness of M^ρ_n for all n. For each n, ensuring fairness of M^ρ_n is the second case in Theorem 7.4, with A_i equal to C_n, which equals $Send_n$. The assumption $\models S \wedge F \Rightarrow \ldots$ in that case of the theorem will have to be implied by fairness conditions on subactions of $Cmp1$.

7.2 Decomposing and Composing Programs

We think of a program as consisting of multiple components. Most often those components are processes. However, a process of an abstract program need not correspond to a process (or thread) in a coding language. As we saw in Section 1.5, Euclid's algorithm can be viewed as a multiprocess program. In general, any disjunct of a program's next-state action can be thought of as a process, and we may be able to write the next-state action in more than one way as the disjunction of subactions. We now use the term *component* instead of *process* for what is described by a disjunct of the program's next-state action.

In this section, we describe a program as the conjunction of separate programs, each describing one of the program's components. There are two reasons for doing this. The first is that we have the program and want to decompose the problem of verifying its correctness into the simpler tasks of verifying correctness of each of its components. We call this procedure *decomposition* and consider it in Section 7.2.1. The second reason is because the program is implemented using existing components whose correctness has been verified. We want to deduce correctness of the program from correctness of the components, without knowing how the components are implemented. This procedure is called *composition* and is considered in Section 7.2.2.

There is little reason to decompose a program if we are proving its correctness. Decomposition structures the proof in three parts: (1) showing that the program is equivalent to the conjunction of programs describing the components, (2) showing that each of those programs satisfies a property, and (3) showing that the conjunction of those properties implies correctness of the original program. Such a proof can be rewritten as an ordinary correctness proof of the original algorithm by a simple rearrangement of the steps of those three parts. But math provides many ways to structure a proof, and deciding in advance to structure it by decomposition might rule out better proofs.

The one good reason to decompose the verification of a program in this way is that it may make it easier to use a tool to verify correctness. For example, a model checker might be able to verify correctness of individual components but not correctness of the complete program. Decomposition would allow using model checking to perform part of the verification, and then using the results presented here to prove that correctness of the components implies correctness of the entire program. This approach has been applied to a nontrivial example [24], but I don't know of any case in which it has been used in industry.

Composition is useful if an engineer wants to verify correctness of a program that describes a system built using an existing component whose behavior is specified by an abstract program. Up until now, we have described a program by a formula that is satisfied by behaviors in which the program to be implemented, which I will here call the actual program, and its environment are both acting correctly. There was no need for the mathematical description to separate the actual program and its environment, since it makes no difference if an execution is incorrect because the programmer didn't understand what the code would do or what the environment would do. However, if a program is implemented using a component purchased elsewhere, it is important to know if an incorrect behavior is due to an incorrect the implementation of the actual program or in the implementation of the component's specification, which is part of the environment.

For composition, we therefore describe a program with two formulas, formula M describing the correct behavior of the actual program and a formula E describing correct behavior of its environment. These formulas are combined into a single formula, written $E \overset{+}{-\!\!\triangleright} M$, that can be thought of as being true of a behavior iff M is true as long as E is (so M is always true if E is always true). Formula $E \overset{+}{-\!\!\triangleright} M$ is what is called a rely/guarantee description of the program [22].

Currently, implementing actual programs with precisely specified existing components seems likely to arise in practice only for components that are traditional programs that perform a computation and stop; and where execution of the component can be considered to be a single step of the complete program. In that case, there is no need for TLA. As explained in Appendix Section A.4, the safety property of the component can be specified by a Hoare triple; and termination is the only required liveness property. Composition in TLA is needed only if the existing component interacts with its environment in a more complex way that must be described with a more general abstract program. Such reusable, precisely specified components do not seem to exist now. Perhaps someday they will.

The results presented here come from a single paper [2]. The reader is referred to that paper for the proofs. To make reading it easier, much of the notation used here—including the identifiers in formulas—is taken from that paper.

7.2.1 Decomposing Programs

7.2.1.1 Writing a Program as a Conjunction

As an example, we take Euclid's algorithm, described in Section 1.5. Instead of x and y, let's name the variables a and b. Also, instead of having the algorithm compute the GCD of two particular numbers, we'll let it nondeterministically choose the initial values of a and b and compute their GCD.

To write the algorithm in TLA, let's first define formulas that describe the initial values of each of the variables, how they are changed in the next-state action, and fairness of the actions that change them, where $\mathbb{N}^+$ is the set of positive integers. Here are those formulas for the variable a:

$$\begin{aligned} \mathit{Init}_a &\triangleq a \in \mathbb{N}^+ \\ \mathit{Next}_a &\triangleq (a > b) \land (a' = a - b) \land (b' = b) \\ L_a &\triangleq \mathrm{WF}_{\langle a \rangle}(\mathit{Next}_a) \end{aligned}$$

The formulas Init_b, Next_b, and L_b are obtained from these formulas by interchanging a and b. Note that L_a is equivalent to $\mathrm{WF}_{\langle a,b \rangle}(\mathit{Next}_a)$ because Next_a implies $b' = b$, and similarly, L_b is equivalent to $\mathrm{WF}_{\langle a,b \rangle}(\mathit{Next}_b)$.

We can describe Euclid's algorithm with the following TLA formula M:

$$\begin{aligned} M &\triangleq \mathit{Init}_M \land \Box[\mathit{Next}_M]_{\langle a,b \rangle} \\ \mathit{Init}_M &\triangleq \mathit{Init}_a \land \mathit{Init}_b \\ \mathit{Next}_M &\triangleq \mathit{Next}_a \lor \mathit{Next}_b \\ L_M &\triangleq L_a \land L_b \end{aligned}$$

Formula M is equivalent to the conjunction of M_a and M_b, defined by:

$$\begin{aligned} M_a &\triangleq \mathit{Init}_a \land \Box[\mathit{Next}_a]_a \land L_a \\ M_b &\triangleq \mathit{Init}_b \land \Box[\mathit{Next}_b]_b \land L_b \end{aligned}$$

The equivalence of M and $M_a \land M_b$ follows by simple logic from:

$$\models \Box[\mathit{Next}_a]_a \land \Box[\mathit{Next}_b]_b \equiv \Box[\mathit{Next}_a \lor \mathit{Next}_b]_{\langle a,b \rangle}$$

This result follows from the equivalence of $\Box(F \land G)$ and $\Box F \land \Box G$, for any formulas F and G, and from

$$[Next_a]_a \;\land\; [Next_b]_b \;\equiv\; [Next_a \lor Next_b]_{\langle a, b \rangle}$$

which follows from the definition of $[\ldots]_{\ldots}$, the equivalence of $(a' = a) \land (b' = b)$ and $\langle a, b \rangle' = \langle a, b \rangle$, and:

$$(7.26) \quad \models Next_a \Rightarrow (b' = b) \qquad \models Next_b \Rightarrow (a' = a)$$

That M is equivalent to $M_a \land M_b$ depends only on (7.26), not on any other properties of $Next_a$ and $Next_b$, and not on the definitions of $Init_a$, $Init_b$, L_a, or L_b. Moreover, it remains true if each variable a and b is replaced by a tuple of variables, as long as those two tuples have no variable in common and (7.26) is satisfied. In other words, if a program consists of two components, each modifying different variables than the other, then the program can be decomposed as the conjunction of two programs, each describing one of the components.

The following theorem generalizes this example from two to n components. It replaces M_a and M_b by processes M_i for $i \in 1 \mathrel{..} n$, replaces a and b by the lists m_i of variables modified by each component i, and replaces (7.26) by the requirement that for each i, the next-state action $Next_i$ of M_i implies $\langle m_j \rangle' = \langle m_j \rangle$ for $j \neq i$.[4]

Theorem 7.5 If $m_1, \ldots, m_n$ are each lists of variables, with all the variables in all the lists distinct, $N \triangleq 1 \mathrel{..} n$, and

$$\begin{aligned}
m \;&\triangleq\; m_1, \ldots, m_n \\
M_i \;&\triangleq\; Init_i \;\land\; \Box[Next_i]_{\langle m_i \rangle} \;\land\; L_i \\
M \;&\triangleq\; \forall\, i \in N : M_i \\
\models M \;&\Rightarrow\; \Box\,[\forall\, i, j \in N \;:\; Next_i \;\land\; (i \neq j) \Rightarrow (\langle m_j \rangle' = \langle m_j \rangle)\,]_m
\end{aligned}$$

then

$$\models M \;\equiv\; (\forall\, i \in N : Init_i) \;\land\; \Box[\exists\, i \in N : Next_i]_{\langle m \rangle} \;\land\; (\forall\, i \in N : L_i)$$

We can further generalize this theorem to allow hiding of variables local to components. Suppose that for each component i, we might want to hide

[4]We are abandoning our convention of naming a list of variables with a boldface letter and adopting the notation of [2], where each m_i is a list of variables that we could write $m_{i,1}, \ldots, m_{i,n_i}$ but won't.

a sublist y_i of the variable list m_i, where the variables y_i do not appear in any M_j with $j \neq i$. We can then define M by

$$M \quad \triangleq \quad \forall\, i \in N \,:\, (\boldsymbol{\exists}\, y_i \,:\, M_i)$$

where if y_i is the empty tuple, then $\boldsymbol{\exists}\, y_i : M_i$ equals M_i. We must modify the hypothesis $\models M \Rightarrow \Box[\ldots]_m$ by replacing m_j with the variables of m_j not in y_j, and make this the conclusion:

$$\begin{aligned} \models M \;\equiv\; \boldsymbol{\exists}\, y_1, \ldots, y_n \,: \\ &(\forall\, i \in N : \mathit{Init}_i) \;\wedge\; \Box[\exists\, i \in N : \widehat{\mathit{Next}}_i]_{\langle m \rangle} \;\wedge\; (\forall\, i \in N : L_i) \end{aligned}$$

where $\widehat{\mathit{Next}}_i \;\triangleq\; \mathit{Next}_i \wedge \forall\, j \in N \setminus \{i\} : y'_j = y_j$.

Another generalization is to require only that different components modify different parts of the state, not necessarily different variables. For example, suppose the components are processes and process i modifies $pc(i)$, where the value of the variable pc is always a function whose domain is the set N. In the hypothesis $\models M \Rightarrow \Box[\ldots]_m$, we can replace the expression $\langle m_j \rangle' = \langle m_j \rangle$ by a state predicate ν_j, where the predicates ν_i must satisfy only the additional hypothesis:

$$\models M \Rightarrow \Box[(\forall\, i \in N \,:\, \nu_i) \Rightarrow (m' = m)]_{\langle m \rangle}$$

For the example in which process i modifies $pc(i)$, we can define ν_i so it implies $(\exists\, S : pc \in (N \rightarrow S)) \wedge (pc'(i) = pc(i))$.

For simplicity, we consider only decomposing programs as described by Theorem 7.5. The rest of what we say about program decomposition can be generalized to these more general ways to decompose programs.

7.2.1.2 Decomposing Proofs

We now consider decomposing the verification that a program is correct into the verification that its components are correct. Let's start with a program with two variables c and d and two components, each component modifying one of the variables. Let's suppose we have decomposed the program into the conjunction $M^l_c \wedge M^l_d$, and we have decomposed its correctness property into $M_c \wedge M_d$, where M_c and M_d are correctness conditions for the two components. We have to verify:

(7.27) $\models M^l_c \wedge M^l_d \Rightarrow M_c \wedge M_d$

and we'd like to do this by verifying that M^l_c satisfies M_c and that M^l_d satisfies M_d. We can obviously do that if we can verify

$$\models M^l_c \Rightarrow M_c \quad \text{and} \quad \models M^l_d \Rightarrow M_d$$

However, those conditions are unlikely to be true. The component program M_c^l is unlikely to satisfy M_c when used as a component of an arbitrary program. Its correctness will depend upon some property of its environment M_d^l. Similarly, correctness of M_d^l will depend upon some property of M_c^l.

We can obviously reduce verification of (7.27) to verifying

$$\models M_c^l \wedge M_d^l \Rightarrow M_c \quad \text{and} \quad \models M_c^l \wedge M_d^l \Rightarrow M_d$$

but that doesn't reduce the amount of work involved. However, suppose that correctness of M_c^l doesn't depend on its environment being the component M_d^l, but just requires its environment to satisfy the correctness condition M_d of that component, and similarly correctness of M_d^l just requires that the other component satisfies M_c. We would then like to reduce verification of (7.27) to verifying:

$$\text{(7.28)} \quad \models M_d \wedge M_c^l \Rightarrow M_c \quad \text{and} \quad \models M_c \wedge M_d^l \Rightarrow M_d$$

This would reduce the amount of work because M_c and M_d are probably significantly simpler than M_c^l and M_d^l. Can we do that?

Let's consider the following trivial example, where each component initializes its variable to 0 and keeps setting its variable's value to the value of the other component's variable.

$$\text{(7.29)} \quad \begin{aligned} M_c^l &\triangleq (c = 0) \wedge \Box[(c' = d) \wedge (d' = d)]_c \\ &\qquad \wedge \text{WF}_c((c' = d) \wedge (d' = d)) \\ M_d^l &\triangleq (d = 0) \wedge \Box[(d' = c) \wedge (c' = c)]_d \\ &\qquad \wedge \text{WF}_d((d' = c) \wedge (c' = c)) \end{aligned}$$

We take as the correctness condition of each component that its variable always equals 0:

$$M_c \triangleq \Box(c = 0) \quad \text{and} \quad M_d \triangleq \Box(d = 0)$$

Condition (7.28) is satisfied because each component's variable keeps setting its variable to 0 (that is, it can take nothing but stuttering steps) if the other component's variable always equals 0. As we would hope, the correctness condition (7.27) is also satisfied because the program consisting of those two components keeps both c and d always equal to 0.

Now let's replace M_c and M_d by the properties asserting that c and d eventually equal 1:

$$M_c \triangleq \Diamond(c = 1) \qquad M_d \triangleq \Diamond(d = 1)$$

while keeping M_c^l and M_d^l the same. Condition (7.28) is still satisfied because each component eventually sets its variable to 1 if the other component sets its variable to 1. However, (7.27) is not satisfied. Changing the correctness conditions doesn't change the behavior of the program, which is to take nothing but stuttering steps.

We might ask why we can't deduce (7.27) from (7.28) in this example. However, the real question is why we *can* deduce it in the first example. Deducing (7.27) from (7.28) is deducing, from the assumption that correctness of each component implies correctness of the other, that both components are correct. This is circular reasoning, and letting $M_c = M_d = \text{FALSE}$ shows that it allows us to deduce that any program implies FALSE, from which we can deduce that the program satisfies any property.

So, why does (7.28) imply (7.27) in the first case? Why can we deduce that both components leave their variables equal to 0 from the assumption that each component leaves its variable equal to 0 if the other process leaves its variable equal to 0? The reason is that neither process can set its variable to a value other than 0 until the other one does. Stated more generally, we can deduce that both components in a two-component program satisfy their correctness properties if neither component can violate its correctness property until after the other does. So we want to replace (7.28) by:

$$\begin{aligned}(7.30)\quad \models\ \forall\, k \in \mathbb{N} :\ & \\ & (M_d \text{ true through state } k-1) \\ & \quad \wedge\ (M_c^l \text{ true through state } k)\ \Rightarrow\ (M_c \text{ true through state } k)\end{aligned}$$

plus the same condition with c and d interchanged, where F true through state -1 is taken to be true for any property F.

To express (7.30) precisely, we have to say what it means for a property F to be true through state k. If F is a safety property, it means that F is true of the finite behavior $\sigma(0) \to \ldots \to \sigma(k)$, which means it's true of the (infinite) behavior obtained by repeating the state $\sigma(k)$ forever. It follows from Theorems 4.3 and 4.4 that any property F equals $\mathcal{C}(F) \wedge L$ where L is a liveness property such that $\langle \mathcal{C}(F), L \rangle$ is machine closed. By the definition of machine closure (in Section 4.2.2.2), any finite behavior that satisfies $\mathcal{C}(F)$ can be completed to a behavior satisfying $\mathcal{C}(F) \wedge L$, which equals F. Therefore, the only way a behavior can fail to satisfy F through state k is for it not to satisfy $\mathcal{C}(F)$ through state k, so F is true through state k means that $\mathcal{C}(F)$ is true through state k. We should therefore replace M_d, M_c^l, and M_c by $\mathcal{C}(M_d)$, $\mathcal{C}(M_c^l)$, and $\mathcal{C}(M_c)$ in (7.30). For a safety property, true through state k means true if all states i with $i > k$ equal state k, so we can rewrite the resulting condition as:

$$\begin{array}{ll}(7.31) & \models\ \forall\, k \in \mathbb{N} : \\ & \quad \text{(every state after state } k \text{ equals state } k) \ \Rightarrow \\ & \qquad (\,(\mathcal{C}(M_d) \text{ true through state } k-1) \ \wedge\ \mathcal{C}(M_c^l) \ \Rightarrow\ \mathcal{C}(M_c)\,)\end{array}$$

Next, let v be the tuple of all variables in these formulas. We can then replace the assertion "every state ... state k" in (7.31) with "$v' = v$ from state k on". By predicate logic, if k does not appear in R or S, then

$$(\forall\, k : P \Rightarrow (Q \wedge R \Rightarrow S)) \ \equiv\ ((\exists\, k : P \wedge Q) \wedge R \Rightarrow S)$$

We can therefore rewrite (7.31) as follows, abbreviating "true through state" as "tts":

$$\begin{array}{ll}(7.32) & (\exists\, k \in Nat\ :\ (v' = v \text{ from state } k \text{ on}) \wedge (\mathcal{C}(M_d) \text{ tts } k-1)) \\ & \quad \wedge\ \mathcal{C}(M_c^l) \ \Rightarrow\ \mathcal{C}(M_c)\end{array}$$

If we define F_{+v} to equal

$$\exists\, k \in Nat\ :\ (v' = v \text{ from state } k \text{ on}) \wedge (\mathcal{C}(F) \text{ tts } k-1)$$

we can then write (7.32) and the condition obtained from it by interchanging c and d as:

$$\begin{array}{ll}(7.33) & \models\ \mathcal{C}(M_d)_{+v} \wedge \mathcal{C}(M_c^l) \ \Rightarrow\ \mathcal{C}(M_c) \\ & \models\ \mathcal{C}(M_c)_{+v} \wedge \mathcal{C}(M_d^l) \ \Rightarrow\ \mathcal{C}(M_d)\end{array}$$

From (7.33), we can infer:

$$(7.34) \quad \models\ \mathcal{C}(M_c^l) \wedge \mathcal{C}(M_d^l) \ \Rightarrow\ \mathcal{C}(M_c) \wedge \mathcal{C}(M_d)$$

The theorems to be stated require a slightly weaker definition of F_{+v}, which makes the conditions (7.33) stronger (so they still imply (7.34)). Let F_{+v}^{old} be the formula that we have been calling F_{+v}. We now define F_{+v} to equal $F_{+v}^{old} \vee F$. With this definition, (7.33) implies its two conditions also hold with the "$+v$" removed. If F is a safety property, then F_v is a safety property but F_{+v}^{old} usually isn't. In fact, if F is a safety property then F_{+v} equals $\mathcal{C}(F_{+v}^{old})$. In practice, the change should seldom make a difference in (7.34) because we don't expect liveness properties to be useful for proving safety properties, so we wouldn't expect $F \wedge G \Rightarrow H$ to be true for safety properties G and H without $\mathcal{C}(F) \wedge G \Rightarrow H$ also being true.

The formula F_{+v} has been defined semantically. However, to verify (7.33) directly, we have to write $\mathcal{C}(F)_{+v}$ as a formula for a given formula F. It's easy to write $\mathcal{C}(F)$ if F has the usual form $Init \wedge \Box[Next]_w \wedge L$, where w is the tuple of variables in the formulas and L is the conjunction of fairness properties of subactions of $Next$. In that case, the definition of machine closure (Section 4.2.2.2) and Theorem 4.6 (Section 4.2.7) imply $\mathcal{C}(F)$ equals $Init \wedge \Box[Next]_w$. We can then write F_{+v} as follows:

$$\begin{array}{lll} F_{+v} & \triangleq & \mathbf{\exists}\, h : \widehat{Init} \wedge [\widehat{Next}]_{w \circ v \circ \langle h \rangle} \\ \text{where } \widehat{Init} & \triangleq & (Init \wedge (h = 0)) \vee (h = 1) \\ \widehat{Next} & \triangleq & \vee\ (h = 0) \wedge \vee\ (h' = 0) \wedge [Next]_w \\ & & \qquad\qquad\quad\ \vee\ h' = 1 \\ & & \vee\ (h = 1) \wedge (h' = h) \wedge (v' = v) \end{array}$$

While writing F_{+v} is easy enough, we usually don't have to for the same reason that we didn't have to use the $+v$ subscripts in (7.27). Our example has one feature that we didn't use in our generalization—namely, that no single program step can make both M_c and M_d false. Here's how to use that feature in general. For safety properties F and G, define $F \perp G$ to be true of a behavior σ iff for every $k \in \mathbb{N}$, if $F \wedge G$ is true of $\sigma(0) \rightarrow \ldots \rightarrow \sigma(k)$ then $F \vee G$ is true of $\sigma(0) \rightarrow \ldots \rightarrow \sigma(k+1)$. Understanding why the following theorem is true is a good test that you understand the definition of F_{+v}.

Theorem 7.6 If F, G, and H are safety properties, v is a tuple of variables containing all variables of F, and $\models H \Rightarrow (F \perp G)$, then $\models F \wedge H \Rightarrow G$ implies $\models F_{+v} \wedge H \Rightarrow G$.

For our example, in which M_c and M_d are safety properties, $\models \mathcal{C}(M_c^l) \Rightarrow (M_c \perp M_d)$ and $\models \mathcal{C}(M_d^l) \Rightarrow (M_c \perp M_d)$ are true and allow us to remove $+v$ from the conditions (7.33). These properties are true because the example satisfies these two conditions:

- We can express correctness of the program as $M_c \wedge M_d$, where only a step of component c can violate M_c and only a step of component d can violate M_d. This seems to be a requirement for decomposing verification of the program into verification of its components to make sense. For example, mutual exclusion can't be expressed as the conjunction of invariance properties that can each be violated by only one process. I therefore expect that attempting to decompose verification of a mutual exclusion algorithm in this way would complicate the task.

- No step is both a c step and a d step. This condition means that a step in a behavior of the program consists of a step of a single component.

> It is the case if the components are processes in a program written in the kind of pseudocode we have been using, which is modeled after the most popular coding languages. I believe that all the engineers I have worked with find this to be the most natural way to describe concurrent programs.

We have obtained the assumptions we need to deduce (7.34). But we want to verify (7.27), which is (7.34) with the $\mathcal{C}$ operators removed. Since F implies $\mathcal{C}(F)$ for any property F, we can remove the $\mathcal{C}$s from the left-hand side of the implication (7.34). But we need additional assumptions to be able to infer that $M_c^l \wedge M_d^l$ implies the liveness parts of M_c and M_d. Since we know that $M_c^l \wedge M_d^l$ implies $\mathcal{C}(M_c) \wedge \mathcal{C}(M_d)$, we can use the following assumptions to deduce (7.27).

$$(7.35) \quad \models \mathcal{C}(M_d) \wedge M_c^l \Rightarrow M_c \quad \text{and} \quad \models \mathcal{C}(M_c) \wedge M_d^l \Rightarrow M_d$$

Condition (7.33) allows us to assume that other components satisfy their safety conditions when showing that a component satisfies its safety condition. However, (7.35) allows us to use the liveness property of only that component when showing that the component satisfies its liveness requirement. It would be circular reasoning to assume M_d when verifying M_c and assume M_c when verifying M_d. However, we can do it for one of the components. For example, if we show that $\mathcal{C}(M_c) \wedge M_d^l$ implies M_d, we can then assume M_d when showing that M_c^l implies M_c. We can therefore replace $\mathcal{C}(M_d)$ by M_d in (7.35), since $\mathcal{C}(M_d) \wedge M_d$ equals M_d. (But we then can't also replace $\mathcal{C}(M_c)$ by M_c.)

For any decomposition of a program into two components M_c^l and M_d^l with correctness properties M_c and M_d, we have deduced (7.27) from (7.33) and (7.35). The following theorem, which is Theorem 1 of [2], generalizes this to a program with n components M_i^l, each with correctness property M_i. The theorem is first stated, then explained.

Theorem 7.7 (Decomposition Theorem) If for all $i \in 1 \,..\, n$:

1. $\models \forall j \in 1 \,..\, n : \mathcal{C}(M_j) \Rightarrow E_i$
2. (a) $\models \mathcal{C}(E_i)_{+v} \wedge \mathcal{C}(M_i^l) \Rightarrow \mathcal{C}(M_i)$

   (b) $\models E_i \wedge M_i^l \wedge (\forall j \in 1 \,..\, (i-1) : M_j) \Rightarrow M_i$

then $\models (\forall i \in 1 \,..\, n : M_i^l) \Rightarrow (\forall i \in 1 \,..\, n : M_i)$

The theorem's conclusion is the obvious generalization of (7.27). There are two hypotheses for each component i. Let's first ignore hypothesis 1 and take E_i to be the conjunction of all the $\mathcal{C}(M_j)$ for $j \neq i$. Hypothesis 2(a) for component M_i^l then generalizes the first condition of (7.33) for component M_c^l, replacing the correctness condition $\mathcal{C}(M_d)$ of the other component with the conjunction $\mathcal{C}(M_j)$ of all the other components. Hypothesis 2(b) makes the similar generalization of (7.35), where allowing the use of all M_j with $j < i$ in the proof of M_i generalizes the observation we made about being able to weaken one of the conditions of (7.35). Hypothesis 1 generalizes what we did for two components in two ways:

- It allows E_i to be the conjunction of all formulas $\mathcal{C}(M_j)$, including $j = i$. This can obviously be done for hypothesis 2(b). It might seem to turn hypothesis 2(a) into a tautology by conjoining the right-hand side $\mathcal{C}(M_i)$ of the implication to the left-hand side, resulting in circular reasoning. However, the subscript $+v$ turns it from circular reasoning into induction on the number of steps in a finite behavior.

- Instead of using the conjunction of the closures of all the components' correctness properties in hypothesis 2, it allows E_i to be any property implied by that conjunction that is strong enough to satisfy hypothesis 2. I expect E_i will always be a safety property, but it's conceivable that it might not be.

The theorem does not make any assumption about v. That's because if w is the tuple of all variables appearing in the formulas (including in v), then F_{+w} implies F_{+v}. Thus, if hypothesis 2(a) is satisfied for any state function v, then it's satisfied with v equal to the tuple of all variables in the formulas. Letting v equal that tuple produces the weakest (hence easiest to satisfy) hypothesis.

7.2.2 Composing Components

In Section 7.2.1, we decomposed a given program as the conjunction of components. We now assume we are given a collection of components and define the program to be the conjunction of those components. As a tiny example, assume we want to write a program that satisfies the property $\Box(c = 0) \wedge \Box(d = 0)$, and we want to do it by conjoining a c component that implements $\Box(c = 0)$ and a d component that implements $\Box(d = 0)$. We find that someone has written a program M_x^l that satisfies $\Box(x = 0)$

when run in an environment that satisfies $\Box(y = 0)$. So, we decide to write our program as $M^l_c \wedge M^l_d$ where:

$$\begin{aligned} M^l_c &\triangleq (M^l_x \text{ WITH } x \leftarrow c,\ y \leftarrow d) \\ M^l_d &\triangleq (M^l_x \text{ WITH } x \leftarrow d,\ y \leftarrow c) \end{aligned}$$

This silly example captures the most important aspect of specifying components: No real device will satisfy a specification such as $\Box(c = 0)$ when executed in an arbitrary environment. For example, a process will not be able to compute the GCD of two numbers if other processes can at any time arbitrarily change the values of its variables.

We want to deduce that $M^l_c \wedge M^l_d$ implies $\Box(c = 0) \wedge \Box(d = 0)$ from the properties that components c and d satisfy, without knowing what M^l_c and M^l_d are. The property that the c component satisfies is that if its environment satisfies $\Box(d = 0)$ then the component satisfies $\Box(c = 0)$; and d satisfies the same condition with d and c interchanged. The obvious way to express these two properties is $\Box(d = 0) \Rightarrow \Box(c = 0)$ and $\Box(c = 0) \Rightarrow \Box(d = 0)$, but those two properties obviously don't imply $\Box(c = 0) \wedge \Box(d = 0)$. We need to find the right way to express mathematically the condition that a component satisfies the property M if its environment satisfies the property E. We do this by assuming that the condition is expressed by a formula $E \stackrel{+}{\rhd} M$ and figuring out what the definition of $\stackrel{+}{\rhd}$ should be, given the assumption that the definition should make this true:

$$(7.36) \quad \models \Box(d = 0) \stackrel{+}{\rhd} \Box(c = 0) \quad \text{and} \quad \models \Box(c = 0) \stackrel{+}{\rhd} \Box(d = 0) \\ \text{implies} \quad \models \Box(c = 0) \wedge \Box(d = 0)$$

We first ask when we can deduce this:

$$(7.37) \quad \models E_1 \stackrel{+}{\rhd} M_1 \quad \text{and} \quad \models E_2 \stackrel{+}{\rhd} M_2 \quad \text{implies} \quad \models M_1 \wedge M_2$$

The answer lies in Theorem 7.7. Since (7.37) doesn't mention the programs M^l_1 and M^l_2, it should be true if we let those programs equal TRUE. Substituting TRUE for them, the conclusion of Theorem 7.7 for $n = 2$ is $M_1 \wedge M_2$. We define $\stackrel{+}{\rhd}$ so that hypotheses 2(a) and 2(b) are equivalent to $E_i \stackrel{+}{\rhd} M_i$. The definition is:

$$E \stackrel{+}{\rhd} M \;\triangleq\; (\mathcal{C}(E)_{+v} \Rightarrow \mathcal{C}(M)) \wedge (E \Rightarrow M)$$

where v is the tuple of all variables in E and M. The theorem then implies that (7.37) is true if hypothesis 1 is satisfied, that hypothesis asserting:

$$(7.38) \quad \models \mathcal{C}(M_1) \wedge \mathcal{C}(M_2) \Rightarrow E_1 \quad \text{and} \quad \models \mathcal{C}(M_1) \wedge \mathcal{C}(M_2) \Rightarrow E_2$$

These conditions are true for our example, so (7.36) is true.

The conclusion of Theorem 7.7 is $\models M_1 \land M_2$, which asserts that the composition of the components satisfies M_1 and M_2 assuming nothing about its environment. We need a more general theorem whose conclusion is $\models E \overset{+}{\triangleright} M_1 \land M_2$, asserting that the composition satisfies $M_1 \land M_2$ if its environment satisfies E. There is actually a stronger result, asserting that the composition satisfies $\models E \overset{+}{\triangleright} M$ for any property M implied by $E \land M_1 \land M_2$. Here is the theorem, generalized from two to n components. It is Theorem 3 of [2].

Theorem 7.8 (Composition Theorem) If

1. $\models \forall\, i \in 1\,..\,n : \mathcal{C}(E) \land (\forall\, j \in 1\,..\,n : \mathcal{C}(M_j)) \Rightarrow E_i$
2. (a) $\models \mathcal{C}(E)_{+v} \land (\forall\, j \in 1\,..\,n : \mathcal{C}(M_j)) \Rightarrow \mathcal{C}(M)$

   (b) $\models E \land (\forall\, j \in 1\,..\,n : M_j) \Rightarrow M$

then $\models (\forall\, j \in 1\,..\,n : E_j \overset{+}{\triangleright} M_j) \Rightarrow (E \overset{+}{\triangleright} M)$

It's instructive to compare Theorems 7.7 and 7.8. They both make no assumption about v, since letting it equal the tuple of all variables in the formulas yields the weakest hypothesis 2(a). Hypothesis 1 differs only in Theorem 7.8 having the additional conjunct $\mathcal{C}(E)$. This conjunct (which weakens the hypothesis) is expected because, if M is the conjunction of the M_i, then the M in the conclusion of Theorem 7.7 is replaced in Theorem 7.8 by $E \overset{+}{\triangleright} M$.

As we observed for Theorem 7.7, hypothesis 1 of Theorem 7.8 pretty much requires the E_i to be safety properties. However, when applying Theorem 7.8, we can choose to make them safety properties by moving the liveness property of E_i into the liveness property of M_i. More precisely, suppose we write E_i as $E_i^S \land E_i^L$, where E_i^S is a safety property and E_i^L a liveness property such that $\langle E_i^S, E_i^L \rangle$ is machine closed; and we similarly write M_i as $M_i^S \land M_i^L$. We can then replace E_i by E_i^S and M_i by $M_i^S \land (E_i^L \Rightarrow M_i^L)$.[5] This replaces the property $E_i \overset{+}{\triangleright} M_i$ by the stronger property:

$$(7.39)\quad E_i^S \overset{+}{\triangleright} (M_i^S \land (E_i^L \Rightarrow M_i^L))$$

It is stronger because if the environment doesn't satisfy its liveness property E_i^L, then $E_i \overset{+}{\triangleright} M_i$ is satisfied no matter what the component does; but in

[5]By definition of machine closure, $\langle M_i^S, M_i^L \rangle$ machine closed implies $\langle M_i^S, E_i^L \Rightarrow M_i^L \rangle$ is also machine closed, because M_i^L implies $E_i^L \Rightarrow M_i^L$.

that case, (7.39) still requires the component to satisfy its safety property M_i^S if the environment satisfies its safety property E_i^S. The two formulas should be equivalent in practice because machine closure of $\langle E_i^S, E_i^L \rangle$ implies that, as long as the environment satisfies its safety property, the component can't know that the environment's entire infinite behavior will violate its liveness property.

Theorem 7.8 has been explained in terms of M_i being the property satisfied by a component whose description M_i^l we don't know, with M a property we want the composition of the components to satisfy. It can also be applied by letting M_i be the actual component M_i^l and letting M be the composition $\forall\, i \in 1 \,..\, n : M_i^l$ of those components. The theorem then tells us under what environment assumption E the composition will behave properly if each M_i^l behaves properly under the environment assumption E_i. However, there is a problem when using it in this way. To explain the problem, we return to our two components c and d whose composition satisfies $\Box(c = 0) \wedge \Box(d = 0)$.

The definitions M_c^l and M_d^l in (7.29) were written for components c and d intended to be composed with one another. They were not written to describe a component that satisfies its desired property only if the environment satisfies its property. We now want to define them and their environment assumptions E_c and E_d so that:

$$\models (E_c \overset{+}{\rhd} M_c^l) \Rightarrow \Box(c = 0)$$
$$\models (E_d \overset{+}{\rhd} M_d^l) \Rightarrow \Box(d = 0)$$

The definition of M_c^l asserts that the value of d cannot change when the value of c changes (because of the conjunct $d' = d$ in the next-state relation) and d cannot change when c doesn't change (because of the subscript $\langle c, d \rangle$). That's a property of its environment. If we want d to satisfy that property, we should state it in E_c, not inside the definition of M_c^l. So, the definition of M_c^l should be

$$M_c^l \;\triangleq\; (c = 0) \wedge \Box[c' = d]_c \wedge \mathrm{WF}_c(c' = d)$$

and the definition of M_d^l should be similarly changed.

If you recall how we decomposed programs in Section 7.2.1.1, expressed in Theorem 7.5, you will realize that the conjuncts $d' = d$ and $c' = c$ in the original definitions of M_c^l and M_d^l were there so that the next-state action of the conjunction of M_c^l and M_d^l would be the disjunction of their next-state actions. With these new definitions, the next-state action of $M^l_c \wedge M^l_d$ equals

$$\begin{array}{l}[\vee\ (c' = d) \wedge (d' = d) \\ \ \ \vee\ (d' = c) \wedge (c' = c) \\ \ \ \vee\ (c' = d) \wedge (d' = c)\ ]_{\langle c, d \rangle}\end{array}$$

The additional disjunct $(c' = d) \wedge (d' = c)$ describes a step that is performed jointly by the two components. If we want the composition of the two components to allow such steps, then there is no problem. However, components are often processes, and in all the examples we've considered, each step is a step of exactly one process. Program descriptions in which each step is performed by a single component are called *interleaving* descriptions.[6]

Suppose we want to consider only interleaving program descriptions. We would take the approach used in Theorem 7.5 that for each i, there is a list m_i of variables that can be modified only by component M^l_i. It would be nice to let E_i assert that any step that changes a variable other than one of the variables of m_i must leave all the variables of m_i unchanged. However, this is impossible because there are infinitely many variables other than the ones in m_i, and a formula E_i can mention only finitely many of them.

Instead, we modify Theorem 7.8 so its conclusion is:

$$(7.40) \quad \models\ G \wedge (\forall\, j \in 1 \,..\, n : E_j \overset{+}{\rhd} M_j) \ \Rightarrow\ (E \overset{+}{\rhd} G \wedge M)$$

for a property G. We can then apply the theorem with M_i equal to M^l_i, M equal to $\forall\, i \in 1 \,..\, n : M^l_i$, and G the property asserting that if $i \neq j$, then a step can't change both $\langle m_i \rangle$ and $\langle m_j \rangle$. Formula $G \wedge M$ is the interleaving description that is presumably what we intended the composition of the components M^l_i to mean.

In the same way as we could generalize Theorem 7.5, we can replace $\langle m_j \rangle' = \langle m_j \rangle$ in the definition of G by a step predicate ν_j and have G assert that every step must satisfy $\nu_i \vee \nu_j$ for $i \neq j$.

You can write the theorem whose conclusion is (7.40) by yourself. All you have to do is apply Theorem 7.8 substituting $n+1$ for n, letting E_{n+1} equal TRUE, letting M_{n+1} equal G, and replacing M with $G \wedge M$. The hypotheses of Theorem 7.8 after making these substitutions are the hypotheses of the theorem.

[6]An interleaving description is often taken to mean any description of a program's executions as sequences of states and/or events, so by that meaning all TLA program descriptions are interleaving descriptions.

Appendix A

Digressions

A.1 Why Not All Mappings Are Sets

Section 2.3.1 states that a function is a special kind of mapping that is assumed to be a value—meaning that it's a set, although we don't know what its elements are. This raises the question of why we can't simply assume that all mappings are sets. The answer is provided by the following theorem, whose proof is due to Stephan Merz. It asserts that there has to be a mapping that is not a set. Although we could assume that some mappings other than functions are sets, the theorem means that we can't assume all mappings are sets. For simplicity, we let functions be the only mappings that we assume to be sets.

Theorem There exists a mapping that is not a set.

1. SUFFICES: ASSUME: Every mapping is a set.
   PROVE: FALSE

   PROOF: Obvious.

DEFINE M to be the mapping such that $M(S) = S$ if S is a set that is a mapping.

2. $M(S)(U) = S(U)$ for every mapping S and every set U.

   PROOF: Since we are assuming that every mapping S is a set, M is a mapping on the collection of all mappings, and by definition $M(S) = S$ for every mapping S, so $M(S)(U) = S(U)$ for every set U.

DEFINE *Russell* to be the mapping on the collection of all sets that are mappings such that $\mathit{Russell}(S) \triangleq \text{CHOOSE } U : U \neq M(S)(S)$.

3. *Russell* is a mapping such that $Russell(S) \neq M(S)(S)$ for all sets S that are mappings.

   PROOF: The value of any syntactically correct formula is a set, even if its elements are unspecified. Therefore, $M(S)(S)$ is a set, and for any set T there exists a set U such that $U \neq T$. Thus, *Russell* is a mapping such that $Russell(S) \neq M(S)(S)$ for every mapping S.

4. $Russell(S) \neq S(S)$ for all sets S that are mappings.

   PROOF: Substituting S for U in step 2 shows $S(S)$ equals $M(S)(S)$, which by step 3 is unequal to $Russell(S)$.

5. Q.E.D.

   PROOF: Since *Russell* is a mapping, and all mappings are assumed to be sets, substituting *Russell* for S in step 4 proves $Russell(Russell) \neq Russell(Russell)$, which equals FALSE.

A.2 Recursive Definitions of Mappings

The general form of a recursive definition of a mapping M is

(A.1) $M(x) \triangleq Def(x, M)$

For example, the definition (2.30) of the mapping # has this form, where *Def* is defined by:

$$Def(x, M) \triangleq \text{IF } x = \{\} \text{ THEN } 0 \\ \text{ELSE } 1 + M(x \setminus \{\text{CHOOSE } e : e \in x\})$$

To make sure that what we do is mathematically sound, we should define what an arbitrary definition of the form (A.1) actually defines the mapping M to be.

If we want the mapping M to be a function with domain D, the example of the factorial function *fact* in Section 2.4.2 shows that we can define M to equal

$$\text{CHOOSE } M : M = (x \in D \mapsto Def(x, M))$$

This suggests that (A.1) should define $M(x)$ to equal the value of $f(x)$ for some function f, whose domain contains x, that satisfies $f(y) = Def(y, f)$ for all y in its domain. To do that, let's first define

$$fdef(D, f) \triangleq x \in D \mapsto Def(x, f)$$

We could then define $M(x)$ to equal:

(A.2) $(\text{CHOOSE } f : f = \mathit{fdef}(\text{DOMAIN}(f), f))(x)$

However, this doesn't work. For example, consider:

$$\mathit{MFact}(x) \;\triangleq\; \text{IF } x = 0 \text{ THEN } 1 \text{ ELSE } x * \mathit{MFact}(x - 1)$$

We would expect $\mathit{MFact}(x)$ to equal $\mathit{fact}(x)$ for all $x \in \mathbb{N}$. However, suppose $x = 13$ and f is a function whose domain is $\{13\}$. Then $f(13)$ equals $13 * f(12)$. But the value of $f(12)$ is unspecified, so it might equal 1. If $f(12)$ did equal 1 and the CHOOSE operator chose that particular function f in (A.2), then this would define $\mathit{MFact}(13)$ to equal 13 rather than $\mathit{fact}(13)$.

The way to ensure that the function f is chosen so that $f(13)$ equals $\mathit{fact}(13)$ is to require that the domain of f includes all numbers in $0 \,..\, 13$. In general, we must choose the domain of f to be large enough so that $f(y) = \mathit{Def}(y, f)$ for all y in its domain uniquely determines f. If that's the case, we say that $\mathit{fix}(f)$ is true. The precise definition of $\mathit{fix}(f)$ is:

$$\forall\, g : (\forall\, y \in \text{DOMAIN}(f) : g(y) = f(y)) \;\Rightarrow\; (f = \mathit{fdef}(\text{DOMAIN}(f), g))$$

(Note that the definition of fix depends on fdef, whose definition depends on Def.) We can then define (A.1) to mean:

(A.3) $M(x) \;\triangleq\; (\text{CHOOSE } f : (x \in \text{DOMAIN}(f)) \wedge \mathit{fix}(f))(x)$

This defines $M(x)$ for all values x. However, is doesn't necessarily define it to equal $\mathit{Def}(x, M)$. We write a definition of the form (A.1) because we want $M(x)$ to equal $\mathit{Def}(x, M)$ for some values of x, so we have to be able to show that it does. We can do that with the following theorem, which is proved in [16].

Theorem

ASSUME: 1. M is defined by $M(x) \triangleq \mathit{Def}(x, M)$.

2. $\succ$ is a well-founded relation on a set S.

3. For all $x \in S$, the value of $\mathit{Def}(x, M)$ depends only on the values of $M(w)$ for $w \in S$.

4. For all $x \in S$ and functions f and g, if $f(w) = g(w)$ for all $w \in \{v \in S : x \succ v\}$, then $\mathit{Def}(x, f) = \mathit{Def}(x, g)$.

PROVE: $\forall\, x \in S : M(x) = \mathit{Def}(x, M)$

A.3 How Not to Write x'''

Here is an amusing paradox. It's illegal to prime a primed expression, so it's illegal to write x'' or x''' if x is a variable. However, consider this definition:

$$\text{(A.4)}\quad F(n) \;\triangleq\; \text{IF } n = 0 \text{ THEN } x \text{ ELSE } F(n-1)'$$

It apparently defines $F(3)$ to equal x'''. It doesn't. To see why not, let's simplify things by defining F to be a function with domain $\mathbb{N}$:

$$F \;\triangleq\; \text{CHOOSE } f : f = (n \in \mathbb{N} \mapsto \text{IF } n = 0 \text{ THEN } x \text{ ELSE } f(n-1)')$$

In this definition f and n are bound constants, so $f(n-1)$ is a constant expression; and $exp' = exp$ for any constant expression exp. Therefore, this definition of F is equivalent to

$$F \;\triangleq\; \text{CHOOSE } f : f = (n \in \mathbb{N} \mapsto \text{IF } n = 0 \text{ THEN } x \text{ ELSE } f(n-1))$$

which defines $f(n)$ to equal x for all $n \in \mathbb{N}$.

If we defined F as a mapping by (A.4), we would take (A.3) as the meaning of that definition and expand the definition of *fix*. This would show that $F(n)$ equals an expression $(\text{CHOOSE } f : \ldots)(3)$ where the primed expression in "$\ldots$" is a constant. Again, we would obtain $F(n) = x$ for all $n \in \mathbb{N}$.

This example illustrates why we should not write a recursive definition that contains an expression that isn't a constant or a state expression. There's no problem applying a recursively defined mapping to a step expression. It should also be all right to apply one to a temporal logic formula, though I can't imagine why we would want to do that. Recursively defined mappings are used to define the meaning of temporal logic operators, but those are mappings of ordinary math whose definitions contain no primes or temporal operators.

A.4 Hoare Logic

Hoare logic is a science of traditional programs developed by C. A. R. (Tony) Hoare [21]. Programs are described in a coding language, and the logic's goal is to prove properties of concrete programs. However, Hoare intended it also to be applied to abstract programs written in code and used in verifying a concrete program.

In Hoare logic, a program is viewed as a relation between the initial and the final states of its execution. A formula of the logic has the form

$\{P\}S\{Q\}$, where S is a program (written in code) and P and Q are state predicates. This formula asserts that if program S is executed starting in a state in which P is true and the execution terminates, then Q is true in the final state of the execution. The formula $\{P\}S\{Q\}$ is called a Hoare triple, P is called its precondition, and Q is called its postcondition. Hoare logic provides a way of showing that a program S satisfies a Hoare triple.

The following is the Hoare logic rule for a program consisting of the single assignment statement $x := exp$, where x is a variable and exp is an expression:

(A.5) $\models (P \text{ WITH } x \leftarrow exp) \Rightarrow Q$ implies $\{P\}\, x := exp\, \{Q\}$

There are also rules for deriving a Hoare triple for a program from Hoare triples of its components. Here are three such rules:

(A.6) $\{P\}S\{R\}$ and $\{R\}S\{Q\}$ imply $\{P\}\, S\,; T\, \{Q\}$

(A.7) $\{P \wedge R\}S\{Q\}$ and $\{P \wedge \neg R\}T\{Q\}$ imply
$\{P\}$ **if** R **then** S **else** T **end if** $\{Q\}$

(A.8) $\models P \Rightarrow I$ and $\{I \wedge R\}S\{I\}$ and $\models I \wedge \neg R \Rightarrow Q$ imply
$\{P\}$ **while** R **do** S **end while** $\{Q\}$

Such rules decompose the proof of a Hoare triple for any program to proofs of Hoare triples for elementary statements of the language, such as assignment statements.

It was quickly realized that pre- and postconditions are not adequate to describe what a program should do. For example, suppose S is a program to sort an array x of numbers. The obvious Hoare triple for it to satisfy has a precondition asserting that x is an array of numbers and a postcondition asserting that x is sorted. But this Hoare triple is true of a program that simply sets all the elements of the array x to 0. A postcondition needs to be able to state a relation between the final values of the variables and their initial values. Various ways were proposed for doing this, one of them being to allow formulas P and Q to contain constants whose values are the same in the initial and final states. For example, the precondition for a sorting program could assert that the constant $x0$ equals x, and the postcondition could assert that the elements of the array x are a sorted permutation of the elements of $x0$.

Viewing a program as a relation between initial and final states means that it can be described mathematically as a formula of the Logic of Actions.

If we represent the program S as an LA formula, then $\{P\}S\{Q\}$ is the assertion $\models P \land S \Rightarrow Q'$; and the Hoare logic rules follow from rules of LA. For example, when programs are described in LA, $S; T$ is $S \cdot T$, where "$\cdot$" is the action composition operator defined in Section 3.4.1.4. The Hoare Logic rule (A.6) is equivalent to this LA rule:

$$\models P \land S \Rightarrow R' \quad \text{and} \quad \models R \land T \Rightarrow Q' \quad \text{imply} \quad \models P \land (S \cdot T) \Rightarrow Q'$$

The program **if** R **then** S **else** T **end if** is represented by the LA formula $(R \land S) \lor (\neg R \land T)$, and rule (A.7) becomes the propositional-logic tautology:

$$\begin{aligned}&\models (P \land R \land S \Rightarrow Q') \land (P \land \neg R \land S \Rightarrow Q') \Rightarrow \\ &\qquad (P \land ((R \land S) \lor (\neg R \land S)) \Rightarrow Q')\end{aligned}$$

Hoare's rule (A.5) for assignment statements is obtained from LA by representing the statement $x := \mathit{exp}$ as $(x' = \mathit{exp}) \land ((v_{\tilde{x}})' = v_{\tilde{x}})$, where $v_{\tilde{x}}$ is the tuple of all program variables other than x. It is valid because $\models P \land (x' = \mathit{exp}) \land ((v_{\tilde{x}})' = v_{\tilde{x}}) \Rightarrow Q'$ equals $\models (P \text{ WITH } x \leftarrow \mathit{exp}) \Rightarrow Q$ if $v_{\tilde{x}}$ is a tuple containing all variables other than x that appear in P or exp.

Rule (A.8) is a bit tricky because, when executed in a state in which R equals FALSE, the **while** statement leaves all variables unchanged. We can represent that **while** statement by

$$((R \land S)^+ \land \neg R') \lor (\neg R \land (v' = v))$$

where v is the tuple of all program variables and $(\ldots)^+$ is defined in Section 3.4.1.4. With this representation of the **while** statement, (A.8) can be derived from the following rule of LA, where I is any state predicate and A any action:

(A.9) $\models I \land A \Rightarrow I'$ implies $\models I \land A^+ \Rightarrow I'$

The LA definition of a Hoare triple implies that the validity of rule (A.8) is proved by the following theorem:

Theorem A.1 ASSUME: 1. $\models P \Rightarrow I$

2. $\models I \land R \land S \Rightarrow I'$

3. $\models I \land \neg R \Rightarrow Q$

4. v is the tuple of all variables occurring in Q.

PROVE: $\models P \land ((R \land S)^+ \land \neg R') \lor (\neg R \land (v' = v)) \Rightarrow Q'$

pagebreak

1. SUFFICES: ASSUME: $P \land ((R \land S)^+ \land \neg R') \lor (\neg R \land (v' = v))$
   PROVE: Q'

   PROOF: Because assumptions 1–3 have the form $\models \ldots$, proving that they imply a formula $F \Rightarrow G$ proves that they imply $\models (F \Rightarrow G)$, and $F \Rightarrow G$ is proved by assuming F and proving G.

2. CASE $P \land (R \land S)^+ \land \neg R'$

   2.1. $I \land (R \land S)^+ \Rightarrow I'$

   PROOF: By assumption 2 and (A.9), with $R \land S$ substituted for A.

   2.2. $P \land (R \land S)^+ \Rightarrow I'$

   PROOF: By 2.1 and assumption 1.

   2.3. I'

   PROOF: By 2.2 and the step 2 case assumption

   2.4. $\neg R'$

   PROOF: By the step 2 case assumption

   2.5. Q.E.D.

   PROOF: By 2.3, 2.4, and assumption 3 (since $\models F$ implies $\models F'$ for any state predicate F).

3. CASE $P \land \neg R \land (v' = v)$

   PROOF: By assumptions 1 and 3, $P \land \neg R$ implies Q, which by $v' = v$ and assumption 4 implies Q'.

4. Q.E.D.

   PROOF: By the step 1 assumption, steps 2 and 3 cover all possibilities.

A.5 Another Way to Look at Safety and Liveness

This section provides a different view of safety and liveness based on viewing behavior predicates as sets of behaviors. This view was first recognized by Gordon Plotkin. I find that it helps me understand safety and liveness. To understand it, we first need some more math.

A.5.1 Metric Spaces

A metric space M is a set with a distance function δ that assigns a non-negative real number $\delta(p, q)$, called the distance between p and q, to all elements p and q of M. The function δ must satisfy these conditions for all elements p, q, and r of M:

M1. $\delta(p, q) = 0$ iff $p = q$.

M2. $\delta(p, q) = \delta(q, p)$

M3. $\delta(p, q) \leq \delta(p, r) + \delta(r, q)$

Do you see why these conditions imply $\delta(p, q) \geq 0$ for all p and q in M?

The set $\mathbb{R}$ of real numbers is a metric space with $\delta(p, q)$ equal to $|p - q|$, where $|r|$ is the absolute value of the number r, defined by

$$|r| \;\triangleq\; \text{IF } r \geq 0 \text{ THEN } r \text{ ELSE } -r$$

An infinite plane, represented as in analytic geometry by the set $\mathbb{R} \times \mathbb{R}$ of pairs of real numbers, is a metric space with $\delta(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle)$ defined to equal $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. I find that thinking of a metric space M as the set of points in a plane is a good way to visualize the concepts presented here.

For a metric space M, the distance $\widehat{\delta}(p, M)$ from $p \in M$ to a nonempty subset S of M is defined to be the largest number r such that $r \leq \delta(p, q)$ for all $q \in S$. For example, if S is the set of all points $\langle x, y\rangle$ in the plane such that $x < 3$, then $\widehat{\delta}(\langle 4, 7\rangle, S)$ equals 1 because $\widehat{\delta}(\langle 4, 7\rangle, S) > 1$ for all q in S and there are elements q of S such that $\delta(\langle 4, 7\rangle, q)$ is arbitrarily close to 1.

For any metric spaces M and subset S of M, if $p \in S$ then $\widehat{\delta}(p, S) = 0$ because condition M1 implies $\delta(p, p) = 0$. In general, $\widehat{\delta}(p, S) = 0$ for $p \in M$ iff for every $e > 0$ there exists $q \in S$ such that $\delta(p, q) < e$.

The closure operation $\mathcal{C}$ on subsets of a metric space M is defined by letting $\mathcal{C}(S)$ be the set $\{p \in M : \widehat{\delta}(p, S) = 0\}$ of all elements M that are a distance 0 from S. For example, if M is the plane, let OD and CD be the open and closed disks of radius 1 centered at the origin, defined by:

$$\begin{aligned} OD &\triangleq \{p \in M : \delta(p, \langle 0, 0\rangle) < 1\} \\ CD &\triangleq \{p \in M : \delta(p, \langle 0, 0\rangle) \leq 1\} \end{aligned}$$

Both $\mathcal{C}(OD)$ and $\mathcal{C}(CD)$ equal CD.

Theorem A.2 For any subset S of a metric space, $S \subseteq \mathcal{C}(S)$ and $\mathcal{C}(S) = \mathcal{C}(\mathcal{C}(S))$.

Proof: The definition of $\mathcal{C}$ and property M1 imply $S \subseteq \mathcal{C}(S)$ for any set S, which implies $\mathcal{C}(S) \subseteq \mathcal{C}(\mathcal{C}(S))$ for any S. Therefore, to show $\mathcal{C}(S) = \mathcal{C}(\mathcal{C}(S))$, it suffices to assume $p \in \mathcal{C}(\mathcal{C}(S))$ and show $p \in \mathcal{C}(S)$. By definition of $\mathcal{C}$ and $\widehat{\delta}$, we do this by assuming $e > 0$ and showing there exists $q \in S$ with $\delta(q, p) < e$. Because $p \in \mathcal{C}(\mathcal{C}(S))$, there exists $u \in \mathcal{C}(S)$ with $\delta(p, u) < e/2$; and $u \in \mathcal{C}(S)$ implies there exists $q \in S$ with $\delta(q, u) < e/2$. By M2 and M3, this implies $\delta(p, q) < e$. End Proof

As you will have guessed by its name, the operator $\mathcal{C}$ on behavior predicates is a special case of the closure operator $\mathcal{C}$ on metric spaces. But for now, forget about behavior predicates and just think about metric spaces.

A set S that, like CD, equals its closure is said to be *closed*. The following result shows that for any set S, its closure $\mathcal{C}(S)$ is the smallest closed set that contains S.

Theorem A.3 For any subsets S and T of a metric space, if T is a closed set and $S \subseteq T$ then $\mathcal{C}(S) \subseteq T$.

Proof: It follows from the definition of $\mathcal{C}$ that $S \subseteq T$ implies $\mathcal{C}(S) \subseteq \mathcal{C}(T)$, and the definition of a closed set implies $T = \mathcal{C}(T)$. End Proof

For any subset S of a metric space M, the *boundary* of S is defined to be the set of all $p \in M$ with $\widehat{\delta}(p, S) = 0$ and $\widehat{\delta}(p, M \setminus S) = 0$. The boundary of both disks OD and CD is $\{p \in M : \delta(p, \langle 0, 0 \rangle) = 1\}$, the circle of radius 1 centered at the origin. For any metric space M and $S \subseteq M$, any element p of M with $\widehat{d}(p, S) = 0$ that is not in S must be in $M \setminus S$ and therefore must satisfy $\widehat{\delta}(p, M \setminus S) = 0$. This shows that the closure of any set S is the union of S and the boundary of S.

A subset S of a metric space M is said to be *dense* iff $\mathcal{C}(S) = M$. A dense set is one that, for any element p of M, contains p or elements of M arbitrarily close to p. As an example, let's call a finite-digit real number one that can be written in decimal notation with a finite number of digits—for example, 123.5432. The set of all pairs of finite-digit numbers is dense in the plane because any real number can be approximated arbitrarily closely with a finite-digit number. Thus, for any pair of real numbers $\langle x, y \rangle$ and any $e > 0$, we can find a pair of finite-digit numbers $\langle p, q \rangle$ within a distance e of $\langle x, y \rangle$ by choosing p and q such that $|x - p|$ and $|y - q|$ are both less than $e/\sqrt{2}$.

Theorem A.4 Any subset S of a metric space equals $\mathcal{C}(S) \cap D$ for a dense set D.

Proof: Let M be the metric space and let D equal $S \cup (M \setminus \mathcal{C}(S))$. The set D consists of all elements of M except those elements in the boundary of S

that are not in S. It follows from this that $\mathcal{C}(S) \cap D = S$. Since elements in the boundary of S are a distance 0 from S, which is a subset of D, they are a distance 0 from D. Therefore all elements in M are a distance 0 from D, so D is dense. End Proof

What we're interested in is not the distance function δ, but the closure operator $\mathcal{C}$. Imagine that the plane was an infinite sheet of rubber that was then stretched and shrunk unevenly in some way. Define the distance between two points on the original plane to be the distance between them after the plane was deformed. For example, if the plane was stretched to make everything twice as far apart in the y direction but the same distance apart in the x direction, then $\delta(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle)$ would equal $\sqrt{(x_1 - x_2)^2 + (2 * (y_1 - y_2))^2}$. As long as the stretching and shrinking is continuous, meaning that the rubber sheet is not torn, the boundary of a set S in the plane after it is deformed is the set obtained by deforming the boundary of S. This implies that the new distance function produces the same closure operator as the ordinary distance function on the plane.

Topology is the study of properties of objects that depend only on a closure operation, which need not be generated by a metric space. But we are interested in a closure operator that is generated by a particular kind of metric space, and it helps me to think in terms of its distance function.

A.5.2 The Metric Space of Behaviors

Recall the correspondence between predicates and sets satisfying those predicates described in Section 2.2.4 as the basis of Venn diagrams. It defines a correspondence between behavior predicates and sets of behaviors. Comparing Section 4.1.3 and A.5.1 should make you expect there to be a definition of the distance between two behaviors that makes the set of all behaviors a metric space in which, under the correspondence between behavior predicates and sets of behaviors, we have:

- The operator $\mathcal{C}$ on behavior predicates corresponds to the closure operator $\mathcal{C}$ on sets of behaviors.
- Safety predicates correspond to closed sets of behaviors.
- Liveness predicates correspond to dense sets of behaviors.

We now face a problem. We want to describe the collection of all behaviors as a metric space. However, mathematicians describe metric spaces as sets, and the collection of all behaviors isn't a set. I believe that all the results about metric spaces that we need would remain true if metric spaces were arbitrary collections rather than sets, but I haven't checked this. So for the rest of this section, we assume that the collection $\mathbb{V}$ of all values is made a set, as explained in Section 2.2.6.4. This makes the collection of all behaviors also a set.

We're interested in the closure operator on sets of behaviors, which can be the same for many different distance functions. The property of the distance function that provides the closure operator we want is that behaviors with a long prefix in common are close together. More precisely, for two different behaviors σ and τ, define $o(\sigma, \tau)$ to be the largest n such that σ and τ have the same prefix of length $n-1$—that is, the largest value n such that $\forall\, i \in 0 \,..\, (n-1) : \sigma(i) = \tau(i)$. Thus, $o(\sigma, \tau)$ equals 1 iff $\sigma(0) = \tau(0)$ and $\sigma(1) \neq \tau(1)$. (There is no such n iff $\sigma = \tau$, in which case we let $o(\sigma, \tau) = \infty$, where $\infty > i$ for all $i \in \mathbb{N}$.) We get the right closure operator on sets of behaviors if δ satisfies this property:

> For any $e > 0$, there is an $n \in \mathbb{N}$ such that $o(\sigma, \tau) > n$ implies $\delta(\sigma, \tau) < e$, for any behaviors σ and τ.

The simplest choice of δ satisfying this and the properties of a distance function for a metric space is $\delta(\sigma, \tau) \triangleq 1/(1 + o(\sigma, \tau))$ for $\sigma \neq \tau$. (Of course, $\delta(\sigma, \tau) = 0$ if $\sigma = \tau$.)

With the correspondence between the operator $\mathcal{C}$ on behavior predicates and the closure operator on this metric space, Theorems 4.2 and 4.3 of Section 4.1.3 are immediate consequences of Theorems A.2–A.4 of Section A.5.1. I find it more elegant to deduce the results about behavior predicates from the corresponding results about metric spaces than to prove them directly. But that might be because I was educated as a mathematician. Whichever you prefer, I hope that having an alternative way of thinking about safety and liveness helps you understand those concepts.

Appendix B

Proofs

B.1 Invariance Proof of *Increment*

Figure 3.5 in Section 3.4.1.1 defines the initial predicate *Init* and next-state action *Next* for the RTLA formula describing a program having the inductive invariant *Inv*. As described in Section 3.4.1.3, to prove the invariance of *Inv* we had to prove the two conditions of (3.10). As with most programs, the proof of the first condition is simple. Here, we describe the proof of the second condition, which is:

Theorem $Next \land Inv \;\Rightarrow\; Inv'$

If the program were described in TLA instead of RTLA, the disjunct *Stutter* would be removed from the definition of *Next*; and *Next* in the theorem would be replaced by $[Next]_v$, where v is the tuple $\langle x, t, pc \rangle$ of variables. The proof of the theorem would be essentially the same, the only difference being that the action *Stutter* would be replaced everywhere by its second conjunct, which is $v' = v$.

The proof of the theorem is decomposed hierarchically. The first two levels are determined by the logical structure of the theorem. There are two standard ways to decompose the proof of a formula of the form $F \Rightarrow G$:

- Write F in the form $F_1 \lor \ldots \lor F_m$ and prove $F_i \Rightarrow G$ for all i.
- Write G in the form $G_1 \land \ldots \land G_n$ and prove $F \Rightarrow G_j$ for all j.

In this case, we can do both, proving $F \Rightarrow G$ by proving $F_i \Rightarrow G_j$ for all i and j, performing the two decompositions in either order. We can do the first decomposition by writing *Next* as a disjunction (since $(P \lor Q \lor \ldots) \land Inv$

equals $(P \wedge \mathit{Inv}) \vee (Q \wedge \mathit{Inv}) \vee \ldots)$, and we can do the second decomposition because *Inv* is defined to be the conjunction of three formulas. We do the first decomposition first.

Expanding the definitions of *Next* and *PgmStep*, using the rule that $\exists$ distributes over $\vee$ ((2.15) of Section 2.1.9.3), we see that *Next* is equivalent to:

$$\begin{array}{l}(\forall\, p \in \mathit{Procs} : \mathit{aStep}(p) \wedge \mathit{Inv}) \;\vee\; (\forall\, p \in \mathit{Procs} : \mathit{bStep}(p) \wedge \mathit{Inv}) \\ \qquad \vee\; (\mathit{Stutter} \wedge \mathit{Inv})\end{array}$$

Writing each $\forall$ assertion as an ASSUME/PROVE, the top level of the proof is:

1. ASSUME: NEW $p \in \mathit{Procs}$, $\mathit{aStep}(p)$, Inv
   PROVE: Inv'

2. ASSUME: NEW $p \in \mathit{Procs}$, $\mathit{bStep}(p)$, Inv
   PROVE: Inv'

3. ASSUME: $\mathit{Stutter}$, Inv
   PROVE: Inv'

   PROOF: By the definitions of *Stutter*, *Inv*, *TypeOK*, and *NumberDone*, since a stutter *Stutter* step leaves the three variables unchanged, which by definition of *Inv* implies that the value of *Inv* is unchanged.

4. Q.E.D.

   PROOF: By steps 1–3 and the definition of *Next*.

Steps 3 and 4 are simple enough that there is no need to decompose their proofs. You should try to understand why these steps, and the others whose proofs are given here, follow from the facts and definitions mentioned in their proofs. To help you, a little bit of explanation has been added to some of the proofs.

We now have to prove steps 1 and 2. They can both be decomposed using the definition of *Inv* as a conjunction. We consider the proof of step 1. Here is the first level of its decomposition.

1.1. TypeOK'

1.2. $\forall\, i \in \mathit{Procs} \;:\; (pc'(i) = b) \;\Rightarrow\; (t'(i) \leq \mathit{NumberDone}')$

1.3. $x' \leq \mathit{NumberDone}'$

1.4. Q.E.D.
PROOF: By steps 1.1–1.3 and the definition of *Inv*.

Step 1.2 is the most difficult one to prove, so we examine its proof. The standard way to prove a formula of this form is to assume $i \in \mathit{Procs}$ and $pc'(i) = b$ and prove $t'(i) \leq \mathit{NumberDone}'$. So, the first step of the proof should be a SUFFICES step asserting that it suffices to make those assumptions and prove $t'(i) \leq \mathit{NumberDone}'$. Thus far, we have used only the logical structure of the formulas, without thinking about what the formulas mean. We can go no further that way. To write the rest of the proof of step 1.2, we have to ask ourselves why an $\mathit{aStep}(p)$ step starting in a state with *Inv* true produces a state with $t'(i) \leq \mathit{NumberDone}'$ true.

When I asked myself that question, I realized that the answer depends on whether or not i is the process p executing the step. That suggested proving the two cases $i \neq p$ and $i = p$ separately, asserting them as CASE statements. In figuring out how to write those two proofs, I found that both of them required proving $\mathit{NumberDone}' = \mathit{NumberDone}$. Moreover, this was true for the same reason in both cases—namely, that an *aStep* step of any process leaves *NumberDone* unchanged. Therefore, I could prove it once in a single step that precedes the two CASE statements. This produced the following level-3 proof:

1.2.1. SUFFICES: ASSUME: NEW $i \in \mathit{Procs}$, $pc'(i) = b$
PROVE: $t'(i) \leq \mathit{NumberDone}'$
PROOF: Obvious.

1.2.2. $\mathit{NumberDone}' = \mathit{NumberDone}$

1.2.3. CASE $i = p$

1.2.4. CASE $i \neq p$

1.2.5. Q.E.D.
PROOF: By steps 1.2.3 and 1.2.4.

This leaves three steps to prove. Here is the proof of step 1.2.4, which I think is the most interesting one.

1.2.4.1. $(t'(i) = t(i)) \land (pc'(i) = pc(i))$

PROOF: By step 1 (which implies $\mathit{aStep}(p)$ and *Inv*), the step 1.2.4 case assumption, and the definitions of *aStep*, *Inv*, and *TypeOK*, which together imply that the values of $t(i)$ and $pc(i)$ are unchanged. (The definition of *TypeOK* is needed because type correctness is required to deduce this.)

1.2.4.2. $pc(i) = b$

PROOF: By step 1.2.4.1 and the step 1.2.1 assumption $pc'(i) = b$.

1.2.4.3. $t(i) \leq \mathit{NumberDone}$

PROOF: By step 1.2.4.2, the step 1 assumption (which implies *Inv*), and the second conjunct in the definition of *Inv*.

1.2.4.4. Q.E.D.

PROOF: Steps 1.2.4.1, 1.2.4.3, and 1.2.2 imply $t'(i) \leq \mathit{NumberDone}'$, which is the current goal (introduced in step 1.2.1).

The purpose of this example is to illustrate the science of proving correctness of concurrent programs. The program and its proof are very simple.[1] The example shows how proving that a program satisfies a property can be hierarchically decomposed into proving simple mathematical assertions whose proofs require no understanding of why the program works or what it's supposed to do. How this can be done for real abstract programs is an engineering problem that is outside the scope of this book.

B.2 Proof of Theorem 4.2

Theorem 4.2 If F is a property, then $\mathcal{C}(F)$ is a safety property such that $\models F \Rightarrow \mathcal{C}(F)$ and, for any safety property G, if $\models F \Rightarrow G$ then $\models \mathcal{C}(F) \Rightarrow G$.

PROOF: Let F be a property. Extend the definition (3.37) of $\natural$ in the obvious way to finite behaviors ρ so that $\natural\rho$ equals ρ with stuttering steps removed.

1. ASSUME: F is a property
   PROVE: $\mathcal{C}(F)$ is a property

   1.1. SUFFICES: ASSUME: σ is a behavior
                  PROVE: σ satisfies $\mathcal{C}(F)$ iff $\natural\sigma$ does

   PROOF: By definition of a property, it suffices to show that $\mathcal{C}(F)$ is SI. By definition of SI, it suffices to assume σ is a behavior and show σ

[1] To check the level-1 proof, the TLA$^+$ proof checker requires only that step 2 be decomposed to a two-step proof, and that it be told to use two simple facts about the cardinality of finite sets that it easily deduces from a standard library of such facts.

satisfies $\mathcal{C}(F)$ iff $\natural\sigma$ does.

1.2. Assume: σ satisfies $\mathcal{C}(F)$
Prove: $\natural\sigma$ satisfies $\mathcal{C}(F)$

Proof: By definition of $\mathcal{C}$, it suffices to assume ρ is a nonempty finite prefix of $\natural\sigma$ and show it is a prefix of a behavior satisfying F. Since ρ is a prefix of $\natural\sigma$, it equals $\natural\tau$ for some prefix τ of σ, so σ satisfies F implies $\tau \circ \nu$ satisfies F for some behavior ν. Since F is SI and ρ is obtained from τ by removing stuttering steps, $\rho \circ \nu$ also satisfies F, so ρ is a prefix of a behavior satisfying F.

1.3. Assume: $\natural\sigma$ satisfies $\mathcal{C}(F)$
Prove: σ satisfies $\mathcal{C}(F)$

Proof: By definition of $\mathcal{C}$, it suffices to show that any finite nonempty prefix ρ of σ satisfies F, which means showing that ρ is the prefix of a behavior satisfying F. Since $\natural\rho$ is a prefix of $\natural\sigma$, by hypothesis there is a behavior τ such that $(\natural\rho) \circ \tau$ satisfies F. Since F is SI and $\rho \circ \tau$ differs from $(\natural\rho) \circ \tau$ only by stuttering steps, $\rho \circ \tau$ too satisfies F. Thus ρ is the prefix of a behavior satisfying F.

1.4. Q.E.D.

Proof: By steps 1.1–1.3.

2. $\mathcal{C}(F)$ is a safety predicate.

2.1. Suffices: Assume: ρ is a prefix of a behavior that satisfies $\mathcal{C}(F)$.
Prove: $\rho^{\uparrow}$ satisfies $\mathcal{C}(F)$

Proof: By definition of safety.

2.2. Let σ be a behavior such that $\rho \circ \sigma$ satisfies F, and let $\phi(n)$ be the sequence of states consisting of n "copies" of the final state of ρ, for any $n \in \mathbb{N}$, and let $\tau(n)$ equal $\rho \circ \phi(n) \circ \sigma$. Then $\tau(n)$ satisfies F for all $n \in \mathbb{N}$.

Proof: A behavior σ such that $\rho \circ \sigma$ satisfies F exists by the step 2.1 assumption and the definition of $\mathcal{C}$. That $\tau(n)$ satisfies F follows from: (i) $\natural\tau(n)$ equals $\natural(\rho \circ \sigma)$ by definition of $\phi(n)$ and $\tau(n)$, (ii) $\rho \circ \sigma$ satisfies F, and (iii) F is SI.

2.3. Every finite prefix of $\rho^{\uparrow}$ is a finite prefix of $\tau(n)$, for some n.

Proof: By the definitions of $\tau(n)$ and of $\rho^{\uparrow}$.

2.4. Q.E.D.

Proof: By steps 2.2 and 2.3, every finite prefix of $\rho^{\uparrow}$ is a prefix of a behavior satisfying F. By definition of $\mathcal{C}$, this implies ρ satisfies $\mathcal{C}(F)$, which by step 2.1 is what we had to prove.

3. ASSUME: G a safety property and $\models F \Rightarrow G$
   PROVE: $\models \mathcal{C}(F) \Rightarrow G$

   PROOF: It suffices to assume σ is a behavior satisfying $\mathcal{C}(F)$ and prove it satisfies G. Since G is a safety property, it suffices to show that any finite prefix ρ of σ satisfies G. By definition of $\mathcal{C}$, ρ is a prefix of a behavior satisfying F, and therefore by hypothesis satisfying G. Since G is a safety property, this implies ρ satisfies G.

4. Q.E.D.

   PROOF: Steps 1–3 are the assertions of the theorem.

B.3 Proof of Theorem 4.3

Theorem 4.3 Every property F is equivalent to $\mathcal{C}(F) \wedge L$ for a liveness property L.

DEFINE $L \triangleq F \vee \neg\mathcal{C}(F)$

1. F is equivalent to $\mathcal{C}(F) \wedge L$.

   PROOF: By $\models (F \Rightarrow \mathcal{C}(F))$ (from Theorem 4.2) and propositional logic.

2. L is a liveness property.

   2.1. SUFFICES: ASSUME: ρ is a finite behavior.
   PROVE: ρ is a prefix of a behavior τ satisfying L.

   PROOF: By definition of liveness, since L is a property because the operators of propositional logic preserve stuttering insensitivity, and $\mathcal{C}(F)$ is a property by Theorem 4.2.

   2.2. CASE ρ is the prefix of a behavior τ satisfying F.

   PROOF: By definition of L, if τ satisfies F then it satisfies L.

   2.3. CASE ρ is not the prefix of any behavior satisfying F.

   PROOF: By definition of $\mathcal{C}(F)$, if ρ were the prefix of a behavior satisfying $\mathcal{C}(F)$, then it would be the prefix of a behavior satisfying F. The CASE assumption therefore implies that any behavior τ having ρ as a prefix does not satisfy $\mathcal{C}(F)$, so it satisfies $\neg\mathcal{C}(F)$ and therefore satisfies L by definition of L.

   2.4. Q.E.D.

   PROOF: Steps 2.2 and 2.3 cover all possibilities.

3. Q.E.D.

   PROOF: By steps 1 and 2.

B.4 Proof of Theorem 4.4

Theorem 4.4 ASSUME: S a safety property and L a liveness property.
PROVE: $\langle S, L \rangle$ is machine closed iff $\mathcal{C}(S \wedge L) \equiv S$.

1. ASSUME: $\langle S, L \rangle$ is machine closed.
   PROVE: $\mathcal{C}(S \wedge L) \equiv S$

   1.1. $\mathcal{C}(S \wedge L) \Rightarrow S$

   PROOF: Theorem 4.2 implies $\models S \Rightarrow \mathcal{C}(S)$, so $\models S \wedge L \Rightarrow \mathcal{C}(S)$. That theorem also implies $\mathcal{C}(S)$ is a safety property and therefore $\models \mathcal{C}(S \wedge L) \Rightarrow \mathcal{C}(S)$. Since S is a safety property, it equals $\mathcal{C}(S)$, so $\models \mathcal{C}(S \wedge L) \Rightarrow \mathcal{C}(S)$ implies $\mathcal{C}(S \wedge L) \Rightarrow S$.

   1.2. $S \Rightarrow \mathcal{C}(S \wedge L)$

   PROOF: It suffices to assume a behavior σ satisfies S and prove σ satisfies $\mathcal{C}(S \wedge L)$. By definition of machine closure, every finite prefix of σ can be completed to a behavior of $S \wedge L$. By definition of $\mathcal{C}$, this implies σ satisfies $\mathcal{C}(S \wedge L)$.

   1.3. Q.E.D.

   PROOF: By steps 1.1 and 1.2.

2. ASSUME: $\mathcal{C}(S \wedge L) \equiv S$
   PROVE: $\langle S, L \rangle$ is machine closed.

   PROOF: By definition of machine closure, it suffices to assume ρ is a finite behavior satisfying S and prove ρ can be completed to a behavior satisfying $S \wedge L$. By definition of what it means for a finite behavior to satisfy a property, $\rho^{\uparrow}$ satisfies S. Since $S \equiv \mathcal{C}(S \wedge L)$, behavior $\rho^{\uparrow}$ satisfies $\mathcal{C}(S \wedge L)$. By definition of $\mathcal{C}$, this implies every prefix of $\rho^{\uparrow}$ can be completed to a behavior satisfying $S \wedge L$, and ρ is a prefix of $\rho^{\uparrow}$.

3. Q.E.D.

   PROOF: By steps 1 and 2.

B.5 Proof of Theorem 4.5

Theorem 4.5 ASSUME: $\models (E\,\mathcal{U}\,P) \;\equiv\; (\Diamond P \land E\,\mathcal{U}\,P) \lor (\neg\Diamond P \land \Box E)$
PROVE: $\models ((E\,\mathcal{U}\,P) \leadsto P) \;\equiv\; (\Box E \leadsto P)$.

1. SUFFICES: $\models ((E\,\mathcal{U}\,P) \Rightarrow \Diamond P) \;\equiv\; (\Box E \Rightarrow \Diamond P)$

   PROOF: $F \leadsto G$ is defined to equal $\Box(F \Rightarrow \Diamond G)$.

2. $\models ((E\,\mathcal{U}\,P) \Rightarrow \Diamond P) \;\equiv\; \begin{array}{l}\land\ (\Diamond P \land E\,\mathcal{U}\,P) \Rightarrow \Diamond P \\ \land\ (\neg\Diamond P \land \Box E) \Rightarrow \Diamond P\end{array}$

   PROOF: By the theorem's assumption and the propositional logic tautology $\models ((F \lor G) \Rightarrow H) \;\equiv\; (F \Rightarrow H) \land (G \Rightarrow H)$.

3. Q.E.D.

   PROOF: $\models \Diamond P \land E\,\mathcal{U}\,P \;\Rightarrow\; \Diamond P$ is a propositional logic tautology, which by step 2 implies $\models ((E\,\mathcal{U}\,P) \Rightarrow \Diamond P) \;\equiv\; (\neg\Diamond P \land \Box E \Rightarrow \Diamond P)$, which with the propositional logic tautology $\models (\neg\Diamond P \land \Box E \Rightarrow \Diamond P) \;\equiv\; (\Box E \Rightarrow \Diamond P)$ proves the step 1 goal.

B.6 Proof of Theorem 4.6

We will need this lemma to prove the theorem:

Lemma B.1 For any countable set I, there is a function f in $\mathbb{N} \to I$ such that for every $i \in I$, there are infinitely many $n \in \mathbb{N}$ with $f(n) = i$

PROOF: Let S_n equal $\{\langle n, i\rangle : i \in I\}$, which is countable because I is. Since $\mathbb{N} \times I$ equals $\bigcup\{S_n : n \in \mathbb{N}\}$, Theorem 2.1 of Section 2.2.6.1 implies $\mathbb{N} \times I$ is countable. Choose g in $\mathbb{N} \to \mathbb{N} \times I$ such that $n \leftrightarrow g(n)$ is a 1-1 correspondence between $\mathbb{N}$ and $\mathbb{N} \times I$. For every $i \in I$, the pair $\langle m, i\rangle$ is in $\mathbb{N} \times I$ for all $m \in \mathbb{N}$, so there are infinitely many $n \in \mathbb{N}$ such that $g(n)(2) = i$. Therefore, $n \in \mathbb{N} \mapsto g(n)(2)$ is the required function f.

Theorem 4.6 Let *Init* be a state predicate, *Next* an action, and v a tuple of all variables occurring in *Init* and *Next*. If A_i is a subaction of *Next* for all i in a countable set I, then the pair

$$\langle\ \mathit{Init} \land \Box[\mathit{Next}]_v\,,\ \forall\, i \in I\ :\ \mathrm{XF}^i_v(A_i)\ \rangle$$

is machine closed, where each XF^i_v may be either WF or SF.

DEFINE $S \;\triangleq\; \mathit{Init} \land \Box[\mathit{Next}]_v$

1. SUFFICES: ASSUME: ρ is a finite behavior satisfying S.
   PROVE: There exists a behavior σ having ρ as a prefix that satisfies S and $\mathrm{SF}_v(A_i)$, for all $i \in I$.

   PROOF: By definition of machine closure, it suffices to show that any finite behavior ρ satisfying S is the prefix of a behavior σ satisfying $S \wedge \forall\, i \in I : \mathrm{XF}^i_v(A_i)$. Since $\mathrm{SF}_v(A_i)$ implies $\mathrm{WF}_v(A_i)$, it suffices to show that σ satisfies $\mathrm{SF}_v(A_i)$ for all $i \in I$.

2. Choose $f \in (\mathbb{N} \rightarrow I)$ such that each $i \in I$ equals $f(n)$ for infinitely many $n \in \mathbb{N}$. Define τ_j for each $j \in \mathbb{N}$ as follows. Let $\tau_0 = \rho$, and for $j > 0$, define τ_j such that:

   if τ_{j-1} is a prefix of a finite behavior μ satisfying S and ending in a state in which $\langle A_{f(j-1)} \rangle_v$ is enabled.
   then τ_j equals the finite behavior obtained by appending to μ a state that makes the last step of τ_j an $\langle A_{f(j-1)} \rangle_v$ step.
   else τ_j is obtained from τ_{j-1} by adding a stuttering step.

   For all $j \in \mathbb{N}$, τ_j is a prefix of and shorter than τ_{j+1}, and τ_j satisfies S.

   PROOF: Lemma B.1 shows the existence of f. By construction each τ_j is a prefix of and shorter than τ_{j+1}. We must just show that τ_j satisfies S. The proof is by induction. It is true for $j = 0$ since τ_0 equals ρ, which satisfies S by the step 1 assumption. So we complete the proof by assuming $j > 0$ and τ_{j-1} satisfies S and proving as follows that τ_j satisfies S.

   If the **if** condition in the definition of τ_j is true, then τ_j satisfies S because τ_{j-1} does and τ_j is obtained by appending to τ_{j-1} an $\langle A_{f(j-1)} \rangle_v$ step, which is a *Next* step by the hypothesis that $\models A_i \Rightarrow \mathit{Next}$ for all $i \in I$. If the **if** condition is false, then τ_j satisfies S because S is stuttering insensitive, τ_{j-1} satisfies S, and τ_j is obtained by adding a stuttering step to τ_{j-1}.

3. Let σ be the behavior having every τ_j as a prefix. Then σ is a behavior satisfying S having the prefix ρ.

   PROOF: The behavior σ exists (and is unique) because each τ_j is a prefix of and shorter than τ_{j+1}. Since ρ equals τ_0, it is a prefix of σ. Step 2 asserts that every prefix τ_j of σ satisfies S. By definition of S, this implies σ satisfies S.

4. σ satisfies $\mathrm{SF}_i(A)$ for all $i \in I$.

   PROOF: Step 2 asserts that for any $i \in I$, there are infinitely many $j \in \mathbb{N}$

such that $i = f(j-1)$. If the **if** condition in the definition of τ_j is true for all of those values of j, then σ contains an $\langle A_i \rangle_v$ step for each of those values of j, so σ satisfies $\Box\Diamond\langle A_i \rangle_v$. If the **if** condition is false for any such j, then τ_{j-1} cannot be extended to any finite behavior containing an $\langle A_i \rangle_v$ step. Hence, $\Box\neg\mathbb{E}\langle A_i \rangle_v$ is true for the suffix of σ obtained by removing the prefix τ_{j-1}. Therefore, σ satisfies $\Diamond\Box\neg\mathbb{E}\langle A_i \rangle_v$, which equals $\neg\Box\Diamond\mathbb{E}\langle A_i \rangle_v$. By (4.14), in either case σ satisfies $\mathrm{SF}_v(A_i)$.

5. Q.E.D.

   PROOF: The theorem follows from steps 1, 3, and 4.

B.7 Proof of Theorem 4.7

Theorem 4.7 Let A_i be an action for each $i \in I$, let $Q \triangleq \exists\, i \in I : A_i$, and let XF be either WF or SF. Then

$$\begin{aligned}\models\ & (\forall\, i \in I : \Box(\mathbb{E}\langle A_i \rangle_v \wedge \Box[\neg A_i]_v \Rightarrow \\ & \qquad\qquad \Box[\neg Q]_v \wedge \Box(\mathbb{E}\langle Q \rangle_v \Rightarrow \mathbb{E}\langle A_i \rangle_v))) \\ & \quad \Rightarrow\ (\mathrm{XF}_v(Q) \equiv \forall\, i \in I : \mathrm{XF}_v(A_i))\end{aligned}$$

DEFINE $\boxtimes\boxtimes$ to equal $\Diamond\Box$ if XF is WF, and $\Box\Diamond$ if XF is SF.

1. SUFFICES: ASSUME: $\forall\, i \in I : \Box(\mathbb{E}\langle A_i \rangle_v \wedge \Box[\neg A_i]_v \Rightarrow \Box[\neg Q]_v \wedge \Box(\mathbb{E}\langle Q \rangle_v \Rightarrow \mathbb{E}\langle A_i \rangle_v))$
   PROVE: $\mathrm{XF}_v(Q) \equiv \forall\, i \in I : \mathrm{XF}_v(A_i)$

   PROOF: Obvious.

2. $\mathrm{XF}_v(Q) \Rightarrow \forall\, i \in I : \mathrm{XF}_v(A_i)$

   2.1. SUFFICES: ASSUME: $\mathrm{XF}_v(Q)$, NEW $i \in I$, $\boxtimes\boxtimes\,\mathbb{E}\langle A_i \rangle_v \wedge \Diamond\Box[\neg A_i]_v$
   PROVE: FALSE

   PROOF: By (4.14) and (4.22), since $F \Rightarrow G$ is equivalent to $F \wedge \neg G \Rightarrow$ FALSE and $\neg\Box\Diamond\langle A_i \rangle_v$ equals $\Diamond\Box[\neg A_i]_v$.

   2.2. $\Diamond(\Box[\neg Q]_v \wedge \Box(\mathbb{E}\langle Q \rangle_v \Rightarrow \mathbb{E}\langle A_i \rangle_v))$

   PROOF: $\boxtimes\boxtimes\,\mathbb{E}\langle A_i \rangle_v$ implies $\Box\Diamond\mathbb{E}\langle A_i \rangle_v$, and $\Box\Diamond\mathbb{E}\langle A_i \rangle_v \wedge \Diamond\Box[\neg A_i]_v$ implies $\Diamond(\mathbb{E}\langle A_i \rangle_v \wedge \Box[\neg A_i]_v)$ by temporal reasoning. Therefore, the step 2.1 assumption implies $\Diamond(\mathbb{E}\langle A_i \rangle_v \wedge \Box[\neg A_i]_v)$, which by the step 1 assumption implies 2.2.

   2.3. $\Diamond\Box[\neg Q]_v \wedge \Diamond\Box(\mathbb{E}\langle Q \rangle_v \equiv \mathbb{E}\langle A_i \rangle_v)$

   PROOF: By step 2.2, because $\Diamond(\Box F \wedge \Box G)$ equals $\Diamond\Box F \wedge \Diamond\Box G$ for any F and G, and $\Box(\mathbb{E}\langle Q \rangle_v \Rightarrow \mathbb{E}\langle A_i \rangle_v)$ implies $\Box(\mathbb{E}\langle Q \rangle_v \equiv \mathbb{E}\langle A_i \rangle_v)$

because the definition of Q implies $\models A_i \Rightarrow Q$, which by the definition of $\mathbb{E}$ in Section 4.2.1 implies $\models \mathbb{E}\langle A_i \rangle_v \Rightarrow \mathbb{E}\langle Q \rangle_v$.

2.4. $\Box\Diamond\langle Q \rangle_v$

PROOF: $\boxdiamond\boxdiamond F$ and $\Diamond\Box(F \equiv G)$ imply $\boxdiamond\boxdiamond G$, for any F and G. Therefore step 2.3 and the step 2.1 assumption $\boxdiamond\boxdiamond\, \mathbb{E}\langle A_i \rangle_v$ imply $\boxdiamond\boxdiamond\, \mathbb{E}\langle Q \rangle_v$. By definition of XF, the step 2.1 assumption $\mathrm{XF}_v(Q)$ and $\boxdiamond\boxdiamond\, \mathbb{E}\langle Q \rangle_v$ imply $\Box\Diamond\langle Q \rangle_v$.

2.5. Q.E.D.

PROOF: Since $\Diamond\Box[\neg Q]_v$ is equivalent to $\neg\Box\Diamond\langle Q \rangle_v$, steps 2.3 and 2.4 imply FALSE, the goal introduced in step 2.1.

3. $(\forall\, i \in I : \mathrm{XF}_v(A_i)) \;\Rightarrow\; \mathrm{XF}_v(Q)$

3.1. SUFFICES: ASSUME: $(\forall\, i \in I \;:\; \mathrm{XF}_v(A_i)) \;\wedge\; \boxdiamond\boxdiamond\, \mathbb{E}\langle Q \rangle_v$
PROVE: $\Box\Diamond\langle Q \rangle_v$

PROOF: By (4.14) and (4.22).

3.2. $\mathbb{E}\langle \mathrm{Q} \rangle_v \Rightarrow \Diamond\langle \mathrm{Q} \rangle_v$

3.2.1. SUFFICES: ASSUME: $(i \in I) \wedge \mathbb{E}\langle A_i \rangle_v$
PROVE: $\Diamond\langle Q \rangle_v$

PROOF: By predicate logic (the $\exists$ Elimination rule), since Q equals $\exists\, i \in I : A_i$, so $\mathbb{E}\langle Q \rangle_v$ equals $\exists\, i \in I : \mathbb{E}\langle A_i \rangle_v$ by rule $\mathbb{E}2$ of Section 5.4.4.2.

3.2.2. CASE: $\Diamond\langle A_i \rangle_v$

PROOF: $\Diamond\langle Q \rangle_v$ follows from the case assumption and $\models A_i \Rightarrow Q$.

3.2.3. CASE: $\Box[\neg A_i]_v$

PROOF: Assumption $\mathbb{E}\langle A_i \rangle_v$ from step 3.2.1, the case assumption, and the step 1 assumption imply $\Box(\mathbb{E}\langle Q \rangle_v \Rightarrow \mathbb{E}\langle A_i \rangle_v)$. This, the temporal logic tautology $\models \Box(F \Rightarrow G) \Rightarrow (\boxdiamond\boxdiamond F \Rightarrow \boxdiamond\boxdiamond G)$, and the step 3.1 assumption $\boxdiamond\boxdiamond\, \mathbb{E}\langle Q \rangle_v$ imply $\boxdiamond\boxdiamond\, \mathbb{E}\langle A_i \rangle_v$. Since $i \in I$ by the step 3.2.1 assumption, the step 3.1 assumption implies $\mathrm{XF}_v(A_i)$, which by $\boxdiamond\boxdiamond\, \mathbb{E}\langle A_i \rangle_v$ and the definition of XF implies $\Diamond\langle A_i \rangle_v$, which by $\models A_i \Rightarrow Q$ implies $\Diamond\langle Q \rangle_v$.

3.3. Q.E.D.

PROOF: All assumptions in effect at step 3.2 are $\Box$ formulas so, as explained in Section 4.2.4, we can deduce $\Box(\mathbb{E}\langle Q \rangle_v \Rightarrow \Diamond\langle Q \rangle_v)$ from 3.2. This and the temporal logic tautology

$$\models \Box(F \Rightarrow \Diamond G) \Rightarrow (\boxdiamond\boxdiamond F \Rightarrow \Box\Diamond G)$$

imply $\boxed{\Diamond}\boxed{\Diamond}\,\mathbb{E}\langle\, Q\,\rangle_v \Rightarrow \Box\Diamond Q$, which by the step 3.1 assumption $\boxed{\Diamond}\boxed{\Diamond}\,\mathbb{E}\langle\, Q\,\rangle_v$ implies the step 3.1 goal $\Box\Diamond Q$.

4. Q.E.D.

   PROOF: By steps 1–3.

B.8 Proof Sketch of Theorem 4.8

Theorem 4.8 Let $\mathbf{x}$ be the list $x_1, \ldots, x_n$ of variables and let F be a property such that $F(\sigma)$ depends only on the values of the variables $\mathbf{x}$ in σ, for any behavior σ. There exists a formula S equal to $\mathit{Init} \wedge \Box[\mathit{Next}]_{\langle \mathbf{x},y \rangle} \wedge \mathrm{WF}_{\langle \mathbf{x},y \rangle}(\mathit{Next})$, where *Init* and *Next* are defined in terms of F, y is a variable not among the variables $\mathbf{x}$, and the variables of S are $\mathbf{x}$ and y, such that $\models F \Rightarrow G$ iff $\models [\![S]\!] \Rightarrow G$, for any property G. If F is a safety property, then the conjunct $\mathrm{WF}_{\langle \mathbf{x},y \rangle}(\mathit{Next})$ is not needed.

PROOF SKETCH: For any behavior σ, let $\sigma|_{\mathbf{x}}$ be the infinite sequence of n-tuples of values such that $\sigma|_{\mathbf{x}}(i)$ equals the value of $\langle \mathbf{x} \rangle$ in state $\sigma(i)$. The basic idea is to define S so that the value of y in any state i of a behavior of S always equals $(\sigma|_{\mathbf{x}})^{+i}$ for some behavior σ satisfying F, and $\mathbf{x}$ always equals $y(0)$. (Remember that τ is the infinite sequence $\tau(0) \rightarrow \tau(1) \rightarrow \cdots$, and τ^{+i} equals $\tau(i) \rightarrow \tau(i+1) \rightarrow \cdots$.)

To do this, for any infinite sequence τ of n-tuples of values, we define $\widetilde{F}(\tau)$ to equal $F(\sigma)$ for any behavior σ such that $\sigma|_{\mathbf{x}}$ equals τ. This uniquely defines $\widetilde{F}$ because, by hypothesis, the value of $F(\sigma)$ depends only on the values of the variables $\mathbf{x}$ in the behavior σ. Define *IsTupleSeq* to be the mapping such that $\mathit{IsTupleSeq}(\tau)$ is true iff τ is an infinite cardinal sequence of n-tuples of arbitrary values. We then then define S by letting:

$$\begin{array}{lcl}
\mathit{Init} & \triangleq & \exists\, \tau : \wedge\ \mathit{IsTupleSeq}(\tau) \ \wedge\ \widetilde{F}(\tau) \\
 & & \qquad\ \ \wedge\ (y = \tau) \ \wedge\ (\langle \mathbf{x} \rangle = \tau(0)) \\
\mathit{Next} & \triangleq & (y' = \mathit{Tail}(y)) \ \wedge\ (\langle \mathbf{x} \rangle' = y'(0))
\end{array}$$

With this definition, $F(\sigma)$ equals TRUE for a behavior σ iff there is a behavior satisfying S in which the initial value of y is $\sigma|_{\mathbf{x}}$. Notice that σ is a halting behavior iff τ ends with an infinite sequence of identical n-tuples. When y equals that value $\mathit{Tail}(y) = y$, so $\langle \mathit{Next} \rangle_{\langle \mathbf{x},y \rangle}$ equals FALSE and S allows only stuttering steps from that point on.

Eliminating the conjunct $\mathrm{WF}_{\langle \mathbf{x},y \rangle}(\mathit{Next})$ allows S to halt even if the behavior y initially equals $\sigma|_{\mathbf{x}}$ for a non-halting behavior σ that satisfies F.

That makes no difference if F is a safety property, since in that case every finite prefix of σ also satisfies F. End Proof Sketch

B.9 Proof of Formula (5.4)

For any variable such as x and any state s, let s_x be the value of x in the state s. For any S-behavior σ:

$$\begin{array}{lll} \multicolumn{3}{l}{[\![(z' = x + y) \text{ WITH } \ldots]\!](\sigma)} \\ \quad\equiv & [\![w' = u + v]\!](\sigma) & \text{By definition of WITH } \ldots. \\ \quad\equiv & \sigma(1)_w = \sigma(0)_u + \sigma(0)_v & \text{By definition of } [\![\ldots]\!]. \\ \multicolumn{3}{l}{[\![z' = x + y]\!](\widehat{f}(\sigma))} \\ \quad\equiv & \widehat{f}(\sigma)(1)_z = \widehat{f}(\sigma)(0)_x + \widehat{f}(\sigma)(0)_y & \text{By definition of } [\![\ldots]\!]. \\ \quad\equiv & f(\sigma(1))_z = f(\sigma(0))_x + f(\sigma(0))_y & \text{By definition of } \widehat{f}. \\ \quad\equiv & \sigma(1)_w = \sigma(0)_u + \sigma(0)_v & \text{By definition of } f. \end{array}$$

B.10 Proof of Theorem 6.2

Theorem 6.2 With the assumptions of Theorem 6.1, for all $i \in I$ let B_i be a subaction of A_i such that

$$(*) \quad T \wedge (i \neq j) \;\Rightarrow\; \Box[\neg(B_i \wedge A_j)]_v$$

for all j in I; and let $B_i^h \triangleq \langle B_i \rangle_v \wedge (h' = \mathit{exp}_i)$. Then

$$T \;\wedge\; (\forall\, i \in I : \mathrm{XF}_v^i(B_i)) \;\;\equiv\;\; \exists\, h : T^h \;\wedge\; (\forall\, i \in I : \mathrm{XF}_{vh}^i(B_i^h))$$

where each XF^i is either WF or SF.

1. Assume: $i \in I$
   Prove: $\langle B_i^h \rangle_{vh} \;\equiv\; \langle B_i \wedge (h' = \mathit{exp}_i) \rangle_v$
   1.1. $\langle B_i^h \rangle_{vh} \;\equiv\; B_i \wedge (v' \neq v) \wedge (h' = \mathit{exp}_i) \wedge (vh' \neq vh)$
      Proof: By the definitions of B_i^h, Next^i, and $\langle \ldots \rangle_{\ldots}$.
   1.2. $\langle B_i^h \rangle_{vh} \;\equiv\; B_i \wedge (v' \neq v) \wedge (h' = \mathit{exp}_i)$
      Proof: By step 1.1, since $vh = v \circ \langle h \rangle$ implies $(v' \neq v) \wedge (vh' \neq vh) \equiv (v' \neq v)$.
   1.3. Q.E.D.

PROOF: By step 1.2 and the definition of $\langle \ldots \rangle_v$.

2. ASSUME: $i \in I$
PROVE: $\mathbb{E}\langle B_i^h \rangle_{vh} \;\equiv\; \mathbb{E}\langle B_i \rangle_v$

PROOF: By step 1 because *exp* is assumed not to contain h', so rules $\mathbb{E}$3 and $\mathbb{E}$5 of Section 5.4.4.2 imply $\mathbb{E}\langle B_i \wedge (h' = exp_i) \rangle_v$ equals $\mathbb{E}\langle B_i \rangle_v$.

DEFINE $\boxed{\Diamond}\boxed{\Diamond}^i$ to equal $\Diamond\Box$ if XF^i is WF and to equal $\Box\Diamond$ if XF^i is SF.

3. ASSUME: $T^h \wedge \forall\, i \in I : \mathrm{XF}^i_{vh}(B_i^h)$
PROVE: $T \wedge \forall\, i \in I : \mathrm{XF}^i_v(B_i)$

3.1. SUFFICES: ASSUME: $(i \in I)$
PROVE: $\mathrm{XF}^i_v(B_i)$

PROOF: By Theorem 6.1, T^h implies T. Therefore, if suffices to prove $\forall\, i \in I : \mathrm{XF}^i_v(B_i)$ to prove step 3

3.2. SUFFICES: ASSUME: $\boxed{\Diamond}\boxed{\Diamond}^i\,\mathbb{E}\langle B_i \rangle_v$
PROVE: $\Box\Diamond\langle B_i \rangle_v$

PROOF: By (4.14) and (4.22)

3.3. $\boxed{\Diamond}\boxed{\Diamond}^i\,\mathbb{E}\langle B_i^h \rangle_{vh}$

PROOF: By the step 3.2 assumption and step 2. (Since step 2 is not in the scope of any assumptions, it implies $\Box(\mathbb{E}\langle B_i^h \rangle_{vh} \;\equiv\; \mathbb{E}\langle B_i \rangle_v)$.)

3.4. $\Box\Diamond\langle B_i^h \rangle_{vh}$

PROOF: The step 3 assumption implies $\mathrm{XF}^i_{vh}(B_i^h)$, which by step 3.3, (4.14), and (4.22) implies $\Box\Diamond\langle B_i^h \rangle_{vh}$.

3.5. Q.E.D.

PROOF: Step 3.4 and step 1 imply $\Box\Diamond\langle B_i \wedge (h' = exp_i) \rangle_v$, which implies the goal introduced in step 3.2.

4. ASSUME: $T \wedge \forall\, i \in I : \mathrm{XF}^i_v(B_i)$
PROVE: $\exists\, h : T^h \wedge \forall\, i \in I : \mathrm{XF}^i_{vh}(B_i^h)$

4.1. SUFFICES: ASSUME: $T^h \wedge (i \in I)$
PROVE: $\mathrm{XF}^i_{vh}(B_i^h)$

PROOF: Theorem 6.1 shows that T implies $\exists\, h : T^h$. This implies that to prove $T \wedge F$ implies $\exists\, h : (T^h \wedge G)$ for any F and G, it suffices to prove that $T \wedge F \wedge T^h$ implies G. Thus, the step 4 assumption shows that to prove the step 4 goal, it suffices to prove T^h implies $\forall\, i \in I : \mathrm{XF}^i_{vh}(B_i^h)$, which is asserted by this step's ASSUME/PROVE.

4.2. SUFFICES: ASSUME: $\boxed{\Diamond}\boxed{\Diamond}^i\,\mathbb{E}\langle B_i^h \rangle_{vh}$

PROVE: $\Box\Diamond\langle B_i^h \rangle_{vh}$

PROOF: By (4.14) and (4.22).

4.3. $⧈⧈^i \mathbb{E}\langle B_i \rangle_v$

PROOF: By the step 4.2 assumption and step 2.

4.4. $\Box\Diamond\langle B_i \rangle_v$

PROOF: The step 4 assumption implies $\mathrm{XF}_v^i(B_i)$, which by step 4.3, (4.14), and (4.22) implies $\Box\Diamond\langle B_i \rangle_v$.

4.5. Q.E.D.

PROOF: By hypothesis, B_i is a subaction of A_i. By hypothesis $(*)$ and the step 4 assumption (which implies T), $B_i \wedge A_j$ equals FALSE if $i \neq j$. Hence a B_i step must be an A_i^h step. By definition of A_i^h and the step 4.1 assumption (which implies T^h), every B_i step is a $B_i \wedge (h' = exp_i)$ step. Therefore, step 4.4 and step 1 imply $\Box\Diamond\langle B_i^h \rangle_{vh}$, which is the goal introduced by step 4.2.

5. Q.E.D.

PROOF: Steps 3 and 4 imply that $T \wedge (\forall\, i \in I : \mathrm{XF}_v^i(B_i))$ is equivalent to $\boldsymbol{\exists}\, h : T^h \wedge (\forall\, i \in I : \mathrm{XF}_{vh}^i(B_i^h))$.

B.11 Proof Sketch of Theorem 6.3

Theorem 6.3 Let T equal $Init \wedge \Box[Next]_{\langle \mathbf{x} \rangle}$ where $\mathbf{x}$ is the list of all variables of S; let F be a safety property such that $F(\sigma)$ depends only on the values of the variables $\mathbf{x}$ in σ, for any behavior σ; and let h be a variable not one of the variables $\mathbf{x}$. We can add h as a history variable to T to obtain T^h and define a state predicate I_F in terms of F such that $\models [\![T]\!] \Rightarrow F$ is true iff I_F is an invariant of T^h.

PROOF SKETCH: Define T^h as follows:

$$\begin{aligned}
T^h \;&\triangleq\; Init^h \wedge \Box[Next^h]_{\langle \mathbf{x}, h \rangle} \\
Init^h \;&\triangleq\; Init \wedge (h = \langle \mathbf{x} \rangle) \\
Next^h \;&\triangleq\; Next \wedge (h' = Append(h, \langle \mathbf{x} \rangle'))
\end{aligned}$$

Since h is a history variable, so $\boldsymbol{\exists}\, h : T^h$ is equivalent to T, to every behavior σ satisfying T there is a corresponding behavior satisfying T^h that has the same values of the variables of $\mathbf{x}$ as σ. Since F does not depend on h, this means that $\models [\![T]\!] \Rightarrow F$ iff $\models [\![T^h]\!] \Rightarrow F$.

We now define $\rho|_x$ and $\widetilde{F}$ to be the same as in the proof of Theorem 4.8, except for finite behaviors ρ. We define $\rho|_{\mathbf{x}}$ to be the finite cardinal sequence of n-tuples of the same length as ρ such that $\rho|_{\mathbf{x}}(i)$ equals the value of $\langle \mathbf{x} \rangle$ in state $\rho(i)$; and we define $\widetilde{F}$ to be the predicate on finite sequences of n-tuples of values such that $\widetilde{F}(\rho|_x)$ equals $F(\rho)$ (which by definition equals $F(\rho^{\uparrow})$). We then define I_F to equal $\widetilde{F}(h)$.

Much as in the proof of Theorem 4.8, for any behavior τ, a finite prefix ρ of τ satisfies F iff $\widetilde{F}(\rho|_{\mathbf{x}})$ is true, which is true iff $\widetilde{F}(h)$ is true of the last state of ρ. Every state of τ is the last state of some finite prefix of τ, and the safety property F is true of τ iff it is true of every finite prefix of τ, so F is true of τ iff I_F is true of every state of τ. This proves that I_F is an invariant of T^h iff T^h satisfies F; and T satisfies F iff T^h does, because F depends only on the variables $\mathbf{x}$. END PROOF SKETCH

B.12 Proof Sketch of Theorem 6.6

Theorem 6.6 Let $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ be lists of variables, all distinct from one another; let the variables of T be $\mathbf{x}$ and $\mathbf{z}$ and the variables of *IS* be $\mathbf{x}$ and $\mathbf{y}$; and let T equal $\mathit{Init} \wedge \Box[\mathit{Next}]_{\langle \mathbf{x},\mathbf{z} \rangle} \wedge L$. Let the operator Φ map behaviors satisfying T to behaviors satisfying *IS* such that $\Phi(\sigma) \sim_y \sigma$. By adding history, stuttering, and prophecy variables to T, we can define a formula $T^{\mathbf{a}}$ such that $\boldsymbol{\exists}\, \mathbf{a} : T^{\mathbf{a}}$ is equivalent to T and a tuple $\mathbf{exp}$ of expressions defined in terms of Φ and the variables of $T^{\mathbf{a}}$ such that

$$\models T^{\mathbf{a}} \Rightarrow (\mathit{IS} \text{ WITH } \mathbf{y} \leftarrow \mathbf{exp})$$

PROOF SKETCH: We first add an infinite stuttering insensitivity t to T, defining T^t to equal

$$\begin{aligned} &\wedge\ \mathit{Init} \wedge \Box[(\mathit{Next} \wedge (v' \neq v)) \vee ((t' \neq t) \wedge (\langle \mathbf{x},\mathbf{z} \rangle' = \langle \mathbf{x},\mathbf{z} \rangle))]_{\langle \mathbf{x},\mathbf{z},t \rangle} \\ &\wedge\ L \wedge \Box\Diamond\langle t' \neq t \rangle_t \end{aligned}$$

In addition to handling the weird case described in Section 6.3.5, this simplifies the proof by not having to consider terminating behaviors, since T^t doesn't allow them.

Let m be the number of variables in $\mathbf{x}$, $\mathbf{z}$, and t. We next define T^{th} by adding to T^t a history variable h whose value in any state is a cardinal sequence of m-tuples. The initial value of h is the one-element sequence whose single element is the value of $\langle \mathbf{x},\mathbf{z},t \rangle$ in the initial state. Each step that does not leave $\langle \mathbf{x},\mathbf{z},t \rangle$ unchanged appends to h the new value of $\langle \mathbf{x},\mathbf{z},t \rangle$. For a behavior σ satisfying T^{th} that has no stuttering steps (steps leaving

$\mathbf{x}$, $\mathbf{z}$, and t unchanged), the value of variable h always equals the sequence $h(0), \ldots, h(Len(h) - 1)$ of m-tuples such that each $h(i)$ equals the value of $\langle \mathbf{x}, \mathbf{z}, t \rangle$ in state $\sigma(i)$ for all $i < Len(h)$.

We now add to T^{th} a prophecy variable p that always makes an infinite sequence of prophecies. The value of p in state $\sigma(i)$ of a behavior σ satisfying T^{thp} is the infinite sequence of m-tuples such that in state $\sigma(i)$, the value of $p(0)$, the next prediction made by p, is the m-tuple that equals the next value of $\langle \mathbf{x}, \mathbf{z}, t \rangle$ different from its current value. This is a generalized prophecy variable with predictions in the collection (not a set) of m-tuple of values. It is defined by writing the next-state action $Next^{th}$ as $\exists\, i : A_i$, where

$$A_i \;\triangleq\; (i = \langle \mathbf{x}, \mathbf{z}, t \rangle') \wedge Next^{th}$$

and defining the next-state action $Next^{thp}$ of T^{thp} by

$$Next^{thp} \;\triangleq\; A_{p(0)} \wedge Next^{th} \wedge (p' = Tail(p))$$

A $Next^{thp}$ step removes the first element from p and appends that element to the end of h. Therefore, the value of $h \circ p$ remains unchanged throughout any behavior that satisfies T^{thp}. The value of $h \circ p$ during a behavior σ satisfying T^{thp} equals the sequence of values of $\langle \mathbf{x}, \mathbf{z}, t \rangle$ in the entire behavior σ, except that σ may have additional (stuttering) steps that leave $\langle \mathbf{x}, \mathbf{z}, t \rangle$ unchanged.

In any state of a behavior satisfying T^{thp}, the value of $(h \circ p)(Len(h) - 1)$ (the last element in the sequence h of m-tuples of values) is the current value of $\langle \mathbf{x}, \mathbf{z}, t \rangle$. The variables h and p, together with the mapping Φ contain all the information needed to define a refinement mapping under which T^{thp} implements IS. To see how this is done, we need some notation.

For any behavior σ and state expression exp, define $\sigma|_{exp}$ to be the infinite sequence of values such that $(\sigma|_{exp})(i)$ equals the value of exp in state $\sigma(i)$, for all $i \in \mathbb{N}$. Thus $\sigma|_{\langle \mathbf{x}, \mathbf{z}, t \rangle}$ is the sequence of m-tuples of values of $\langle \mathbf{x}, \mathbf{z}, t \rangle$ in the states of σ. Define the mapping $\widetilde{\Phi}$ from m-tuples of values to behaviors so that $\widetilde{\Phi}(\rho)$ equals $\Phi(\sigma)$ for some behavior σ such that $\sigma|_{\langle \mathbf{x}, \mathbf{z}, t \rangle} = \rho$. (It doesn't matter what values the states of σ assign to variables other than those in $\mathbf{x}$, $\mathbf{z}$, and t since they don't affect whether or not σ satisfies T.) We are assuming that $\Phi(\sigma)$ satisfies IS and $\Phi(\sigma) \sim_{\mathbf{y}} \sigma$. Therefore, for any behavior satisfying T^{thp}, for the value of $h \circ p$ in any state of that behavior, $\widetilde{\Phi}(h \circ p)$ satisfies IS and $\widetilde{\Phi}(h \circ p) \sim_{\mathbf{y}} \sigma$ for some behavior σ such that $\sigma|_{\langle \mathbf{x}, \mathbf{z} \rangle} = h \circ p$.

To understand how to construct the needed refinement mapping, we consider a simpler version of the theorem that would be true if we were using

RTLA rather than TLA, so we didn't have the complication introduced by stuttering insensitivity In that case, $\widetilde{\Phi}(h \circ p)$ would satisfy $\widetilde{\Phi}(h \circ p) \simeq_{\mathbf{y}} \sigma$ instead of $\widetilde{\Phi}(h \circ p) \sim_{\mathbf{y}} \sigma$. This means that the behavior $\Phi(\sigma)$ satisfying *IS* is constructed from a behavior σ with $\sigma|_{\langle \mathbf{x}, \mathbf{z}, t \rangle}$ equal to $h \circ p$ by just changing the value of the variables $\mathbf{y}$ in each state of σ (without adding or removing stuttering steps). For any state s, let $s_{\mathbf{y}}$ be the values of the variables $\mathbf{y}$ in that state. For σ to satisfy *IS*, the values of $\mathbf{y}$ in any state $\sigma(i)$ of the behavior σ should equal the values of $\Phi(\sigma)(i)_{\mathbf{y}}$, the values of $\mathbf{y}$ in the corresponding state of $\Phi(\sigma)$.[2] Remember that state number i of σ corresponds to the m-tuple $(h \circ p)(i)$ with $i = Len(h) - 1$. Therefore, the values assigned to $\mathbf{y}$ by the refinement mapping under which T^{thp} implements *IS* should in each state equal the values of $\mathbf{y}$ in state number $Len(h) - 1$ of $\widetilde{\Phi}(h \circ p)$. In other words, we have:

(B.1) $\models T^{thp} \Rightarrow (IS \text{ WITH } \mathbf{y} \leftarrow (\widetilde{\Phi}(h \circ p)(Len(h) - 1))_{\mathbf{y}})$

Let's now return to the actual situation in which $\widetilde{\Phi}(h \circ p) \sim_{\mathbf{y}} \sigma$ (rather than $\widetilde{\Phi}(h \circ p) \simeq_{\mathbf{y}} \sigma$). The behavior $\widetilde{\Phi}(h \circ p)$ is constructed from the behavior σ, where $\sigma|_{\langle \mathbf{x}, \mathbf{z}, t \rangle}$ equals the value of $h \circ p$ in every state of σ, by possibly adding stuttering steps to σ and then modifying the values of $\mathbf{y}$ in its states. (There are no stuttering steps to remove from σ because we defined T^t and the history variable h so that h is a sequence of m-tuples no two successive elements of which are equal.)

To fix (B.1) to handle these stuttering steps, we have to add a stuttering variable s to T^{thp} to obtain the program T^{thps} that adds to the behavior σ satisfying T^{thp} the stuttering steps needed to produce a behavior τ such that the state $\tau(i)$ corresponds to the state $\widetilde{\Phi}(h \circ p)(i)$ for every $i \in \mathbb{N}$. The behavior $\widetilde{\Phi}(h \circ p)$ tells us where those stuttering steps must be added: each step $\widetilde{\Phi}(h \circ p)(i) \rightarrow \widetilde{\Phi}(h \circ p)(i+1)$ that leaves the value of $\langle \mathbf{x}, \mathbf{z}, t \rangle$ unchanged corresponds to a stuttering step added to the behavior σ satisfying T^{thp} to form τ.

Define the function f in $(\mathbb{N} \rightarrow \mathbb{N})$ in terms of $h \circ p$ as follows. Let $f(0) = 0$, and for all $i \in \mathbb{N}$, let $f(i+1)$ be the smallest number greater than $f(i)$ such that the value of $\langle \mathbf{x}, \mathbf{z}, t \rangle$ in state $\widetilde{\Phi}(h \circ p)(f(i+1))$ is different from its value in state $\widetilde{\Phi}(h \circ p)(i)$. (Such a number $f(i+1)$ exists for the value of $h \circ p$ in any reachable state of T^{thp} because adding the infinite-stuttering variable t ensured that $\Box\Diamond(t' \neq t)$ is true for every behavior of T^{thp}.)

To construct $\widetilde{\Phi}(h \circ p)$ from a behavior σ with $\sigma|_{\langle \mathbf{x}, \mathbf{z}, t \rangle} = h \circ p$, we have to add stuttering steps to make each state number i of σ correspond to

[2] This means that for each variable y_j of $\mathbf{y}$, the value of y_j in any state $\sigma(i)$ should equal the value of y_j in $\Phi(\sigma)(i)$.

state number $f(i)$ of $\widetilde{\Phi}(h \circ p)$. The stuttering variable s added to T^{thp} to define T^{thps} has to add $f(i+1) - f(i) - 1$ stuttering steps between states number i and $i+1$ of a behavior of T^{thp}. That means adding $f(Len(h)) - f(Len(h)-1) - 1$ stuttering steps after every step of the next-state action of T^{thp}. (Of course no stuttering step is added if $f(Len(h)) - f(Len(h)-1) = 1$.) We then define k to be a state expression whose value in any behavior satisfying T^{thps} is the number of steps of the next-state action of T^{thps} that have occurred so far. We can define k in terms of the values of f and s, or we can simply add k as a history variable to T^{thps}. However we define it, we can express the corrected version of (B.1) that handles stuttering insensitivity by:

$$\models T^{thps} \Rightarrow (IS \text{ WITH } \mathbf{y} \leftarrow (\widetilde{\Phi}(h \circ p)(k))_{\mathbf{y}})$$

which completes the proof. END PROOF SKETCH

Replacing the Prophecy Variable with a Prophecy Constant

Theorem 6.6 is true if we use a prophecy constant, as defined in Section 6.6, instead of a prophecy variable. This is proved by modifying the proof sketch of Theorem 6.6 as follows. After adding the auxiliary variables t and h, we add to T^{th} a prophecy constant c whose value is the infinite sequence of m-tuples that are the values of $\langle \mathbf{x}, \mathbf{z}, t \rangle$ in the entire behavior. In other words,

$$T^{thc} \triangleq \exists\, c : P(h, c) \wedge T^{th}$$

where $P(h, c)$ asserts that c is an infinite sequence of m-tuples such that, in every state, h equals the sequence of the first $Len(h)$ elements of c. We then define p to be the state function such that, in any state, $h \circ p$ equals c. The proof is then the same as the rest of the proof of Theorem 6.6, using this as the definition of p.

B.13 Proof of Theorem 7.2

The proof uses the following assertion, which the definition of the action composition operator "$\cdot$" implies is true for all actions A_i and B_j:

$$\text{(B.2)} \quad \models (\exists\, i \in I : A_i) \cdot (\exists\, j \in J : B_j) \;\equiv\; (\exists\, i \in I, j \in J : A_i \cdot B_j)$$

Theorem 7.2 If $A \equiv \exists\, i \in I : A_i$ and $B \equiv \exists\, j \in J : B_j$ for actions A_i and B_j, then:

$$\models (\forall\, i \in I, j \in J : A_i \cdot B_j \Rightarrow B_j \cdot A_i) \Rightarrow (A \cdot B \Rightarrow B \cdot A)$$

PROOF: It suffices to assume $A_i \cdot B_j \Rightarrow B_j \cdot A_i$ for all $i \in I$ and $j \in J$ and prove $A \cdot B \Rightarrow B \cdot A$. Here is the proof:

$$\begin{array}{rcll} A \cdot B & \equiv & (\exists\, i \in I,\, j \in J : A_i \cdot B_j) & \text{by (B.2)} \\ & \Rightarrow & (\exists\, i \in I,\, j \in J : B_j \cdot A_i) & \text{we assume } A_i \cdot B_j \Rightarrow B_j \cdot A_i \text{ for all } i \text{ and } j \\ & \equiv & B \cdot A & \text{by (B.2), substituting } I \leftarrow J,\ J \leftarrow I, \\ & & & A_i \leftarrow B_j, \text{ and } B_j \leftarrow A_i. \end{array}$$

END PROOF

B.14 Proof of Theorem 7.3

Theorem 7.3 If S equals $Init \land \Box[Next]_{\mathbf{x}}$, P is a safety property, and Q is a state predicate such that $\models S \Rightarrow \Box\mathbb{E}([Next]_{\mathbf{x}}^{+} \land Q')$, then $\models S \land \Box\Diamond Q \Rightarrow P$ implies $\models S \Rightarrow P$.

1. $\langle S, \Box\Diamond Q \rangle$ is machine closed.

  1.1. SUFFICES: ASSUME: ρ a finite behavior satisfying S
  PROVE: ρ is a prefix of a behavior satisfying $S \land L$

  PROOF: By definition of machine closure.

  1.2. There is a mapping Φ such that if μ is any finite behavior satisfying S, then $\Phi(\mu)$ is a finite behavior ending in a state satisfying Q such that $\mu \circ \Phi(\mu)$ satisfies S.

  PROOF: The hypothesis $\models S \Rightarrow \Box\mathbb{E}([Next]_{\mathbf{x}}^{+} \land Q')$ implies that the action $[Next]_{\mathbf{x}}^{+} \land Q'$ is enabled in the last state of μ. Therefore, there is a finite behavior ψ beginning with the last state of μ and containing at least two states such that the last state of ψ satisfies Q and every step of ψ satisfies $[Next]_{\mathbf{x}}^{+}$. Let $\Phi(\mu)$ equal $Tail(\psi)$. Then $\mu \circ \Phi(\mu)$ satisfies S by definition of S, because μ satisfies $Init$ and $\Box[Next]_{\mathbf{x}}$ and every step of ψ is a $[Next]_{\mathbf{x}}^{+}$ step; and the last state of $\mu \circ \Phi(\mu)$ satisfies Q.

  DEFINE τ_i for $i \in \mathbb{N}$ by: $\tau_0 \triangleq \rho$
  $\tau_{i+1} \triangleq \tau_i \circ \Phi(\tau_i)$ for all $i \in \mathbb{N}$

  1.3. Q.E.D.

  PROOF: Let σ be the (unique) behavior such that each τ_i is a prefix of σ, so ρ (which equals τ_0) is a prefix of σ. Each finite prefix of σ is a prefix of some τ_i and therefore satisfies S. Since S is a safety property, this implies σ satisfies S. For all $i \in \mathbb{N}$, τ_{i+1} has at least one more state satisfying Q than τ_i does, so σ satisfies $\Box\Diamond Q$.

2. $S = \mathcal{C}(S \wedge \Box\Diamond Q)$

   Proof: By step 1 and Theorem 4.4.

3. Q.E.D.

   Proof: By step 2, the assumption that P is a safety property, and Theorem 4.2.

Bibliography

[1] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994. This paper has an appendix published by ACM only online that contains proofs. Other online versions of the paper might not contain the appendix.

[2] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995. This paper has an appendix published by ACM only online that contains proofs. Online versions of the paper might not contain the appendix.

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

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

[5] Selma Azaiez, Damien Doligez, Matthieu Lemerre, Tomer Libal, and Stephan Merz. Proving determinacy of the PharOS real-time operating system. In Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklós Biró, editors, *5th Intl. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016)*, volume 9675 of *LNCS*, pages 70–85. Springer, 2016.

[6] Arthur Bernstein and Paul K. Harter, Jr. Proving real time properties of programs with temporal logic. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 1–11, New York, 1981. ACM. *Operating Systems Review 15*, 5.

[7] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993.

[8] Ernie Cohen and Leslie Lamport. Reduction in TLA. In David Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998.

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

[10] Thomas W. Doeppner, Jr. Parallel program correctness through refinement. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 155–169. ACM, January 1977.

[11] Laurent Doyen, Goran Frehse, George J. Pappas, and André Platzer. Verification of hybrid systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 1047–1110. Springer, 2018.

[12] Michael Fischer. Re: Where are you? Email message to Leslie Lamport. Arpanet message sent on June 25, 1985 18:56:29 EDT, number 8506252257.AA07636@YALE-BULLDOG.YALE.ARPA (47 lines), 1985.

[13] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

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

[15] Aman Goel, Stephan Merz, and Karem A. Sakallah. Towards an automatic proof of the bakery algorithm. In Marieke Huisman and António Ravara, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 13910 of *Lecture Notes in Computer Science*, pages 21–28. Springer, 2023.

[16] George Gonthier and Leslie Lamport. Recursive operator definitions. Research Report RR-9341, Inria Saclay Ile de France, 2020. `https://inria.hal.science/hal-02598330/file/RR-9341.pdf`.

[17] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What good are digital clocks? In Werner Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna,*

Austria, July 13-17, 1992, Proceedings, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer, 1992.

[18] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, January 1990.

[19] Wim H. Hesselink. Eternity variables to prove simulation of specifications. *ACM Trans. Comput. Log.*, 6(1):175–201, 2005.

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

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

[22] Cliff B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332, Amsterdam, September 1983. IFIP, North-Holland.

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

[24] R. P. Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In Costas Courcoubetis, editor, *Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 166–179, Berlin, June 1993. Springer-Verlag. Proceedings of the Fifth International Conference, CAV'93.

[25] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy caching in TLA. *Distributed Computing*, 12(2/3):151–174, 1999.

[26] Leslie Lamport. The paxos algorithm or how to win a turing award. Web page. `https://lamport.azurewebsites.net/tla/paxos-algorithm.html`.

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

[28] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[29] Leslie Lamport. The mutual exclusion problem—part ii: Statement and solutions. *Journal of the ACM*, 32(1):327–348, January 1986.

[30] Leslie Lamport. Hybrid systems in TLA$^+$. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102, Berlin, Heidelberg, 1993. Springer-Verlag.

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

[32] Leslie Lamport. Proving possibility properties. *Theoretical Computer Science*, 206(1–2):341–352, October 1998.

[33] Leslie Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4):239–245, November 2000.

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

[35] Leslie Lamport. Real-time model checking is really simple. In Dominique Borrione and Wolfgang Paul, editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, volume 3725 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2005.

[36] Leslie Lamport. The PlusCal algorithm language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing, ICTAC 2009*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer-Verlag, 2009.

[37] Leslie Lamport and Stephan Merz. Auxiliary variables in TLA+. arXiv:1703.05121 (`https://arxiv.org/abs/1703.05121`) Also available, together with TLA$^+$ specifications, at `http://lamport.azurewebsites.net/tla/auxiliary/auxiliary.html`.

[38] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reason.*, 40(1):1–33, 2008.

[39] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[40] 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.

[41] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17. ACM Press, August 1988.

[42] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

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

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

[45] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[46] 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.

[47] 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.

[48] Hagen Völzer and Daniele Varacca. Defining fairness in reactive and concurrent systems. *J. ACM*, 59(3):13:1–13:37, 2012.

Index

The index obviously needs to be added.