TLA$^+$ in Practice and Theory
Part 3: The (Temporal) Logic of Actions

08 Jun 2017

This is part 3 in a four-part series. A new post will be published every Thursday. ***Part 1***, ***part 2***

If you find TLA$^+$ and formal methods in general interesting, I invite you to visit and participate in the new ***/r/tlaplus*** *on Reddit.*

In part 2 we learned how we specify data and operations on data. In this post we will finally learn how we specify and reason about *processes*, or *dynamic systems* — in particular, algorithm — in TLA$^+$ and how we check their correctness. While there is only very little syntax left to learn, our study of the theory of representing programs as mathematical objects is now just beginning. Most of our discussion will not concern TLA$^+$ specifically, but TLA, Lamport's Temporal Logic of Actions.

This post is the longest in the series by far. Covering in depth a mathematical theory — even a relatively simple one — for reasoning about computation is not light reading. But I will repeat my warning from part 1: very little of the theory and its background presented here is necessary in order to use TLA$^+$ to specify and verify real-world software. If you do choose to delve into the depths of TLA$^+$'s theory, remember there's a whole week until the final installment in this series, so take your time.

A Bit of Historical Background

The need to mathematically reason about non-trivial programs was recognized very early on in the history of computation. Possibly the earliest attempt at a method for mathematical reasoning about programs — the first formal method — was devised by Alan Turing in his paper *Checking a Large Routine*[1] and presented in a talk in 1949, who, interestingly, used a similar notation, that of primed variables denoting a variable's value following a program step, to that of Lamport's TLA. Unfortunately, it seems that talk, like much of Turing's work, was ignored at the time and probably did not influence later ideas[2]. A very similar, yet somewhat expanded approach to that of Turing's, was invented by Robert Floyd in 1967 and then presented in a more formal way by Tony Hoare in 1969 as what is now known as Floyd-Hoare logic (or just Hoare logic).

Turing's paper begins thus:

> *How can one check a routine in the sense of making sure that it is right?*
>
> *In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.*

Turing's, Floyd's and Hoare's method, uses what's known as "assertional reasoning"; its main idea — listing at each point in the program which facts, assertions, about the program's state[3] are true at that point.

Lamport, who in 1977 began working on expanding Floyd and Hoare's techniques to concurrent programs, was an early believer in the idea of program verification. In 1979 he wrote a letter to the editor of *Communications of the ACM* regarding their recently introduced rules for publishing algorithms:

> *For years, we did not know any better way to check programs than by testing them to see if they worked… But the work of Floyd and others has given us another way. They taught us that a program is a mathematical object, so we can apply the reasoning methods of mathematics to deduce its properties… After Euclid, a theorem could no longer be accepted solely on the basis of evidence provided by drawing pictures. After Floyd, a program should no longer be accepted solely on the basis of how it works on a few test cases. A program with no demonstration of why it is correct is the same as a conjecture — a statement which we think may be a theorem. A conjecture must be exceptionally interesting to warrant publication. An unverified program should also have to be exceptional to be published… The ACM should require that programmers convince us of the correctness of the programs that they publish, just as mathematicians must convince one another of the correctness of their theorems. Mathematicians don't do this by giving "a sufficient variety of test cases to exercise all the main features," and neither should computer scientists.*

One point of contention between Lamport and others who were working on reasoning about programs at the time from a more linguistic point of view, like Tony Hoare and Robin Milner, was about the representation of the program's control state, namely the program counter and (if applicable) the call stack. Those who work on programming language theory are adamantly reluctant — to this day — to make the program's control's state explicit, while Lamport claims that it makes reasoning about concurrency significantly easier.

Amir Pnueli's introduction of temporal logic into computer science was somewhat of a watershed moment for software verification, resulting in not one but two Turing Awards, one to Pnueli and one to Clarke, Emerson and Sifakis for the invention of temporal logic model checkers[4]. Pnueli's logic offered "a unified approach to program verification… which applies to both sequential and parallel programs."[5] Note that "parallel programs" also refers to nonterminating interactive programs, where it's useful to consider the user as a concurrent process.

Lamport was impressed with temporal logic but became disillusioned with its practical application when he saw his colleagues "spending days trying to specify a simple FIFO queue — arguing over whether the properties they listed were sufficient. I realized that, despite its aesthetic appeal, writing a specification as a conjunction of temporal properties just didn't work in practice."

The **Temporal Logic of Actions** (TLA), which forms the core of TLA$^+$, was invented in the late '80s and is the culmination of Lamport's ideas about how to reason about algorithms and software or hardware systems. TLA differs from earlier uses of temporal logics in two important ways. The first is that Lamport, a staunch believer in the power of ordinary math, made TLA restrict the need for temporal reasoning to a bare minimum:

> *TLA differs from other temporal logics because it is based on the principle that temporal logic is a necessary evil that should be avoided as much as possible. Temporal formulas tend to be harder to understand than formulas of ordinary first-order logic, and temporal logic reasoning is more complicated than ordinary mathematical … reasoning.*[6]

Another difference is that temporal logic is usually used to reason about programs written in some programming language, while TLA is a universal mathematical notation in which one writes both the algorithm and its claimed properties as formulas in the same logic. This makes TLA a *program logic*, and a particularly powerful one at that. **Lamport writes**:

> *Classical program verification, begun by Floyd and Hoare, employs two languages. The program is written in a programming language, and properties of the program are written in the language of formulas of some logic. Properties are derived from the program text by special proof rules, and from other properties by reasoning within the logic… A program logic expresses both programs and properties with a single language. Program* Π *satisfies property* P *if and only if* $\Pi \Rightarrow P$ *is a valid formula of the logic.*

Algorithms and Programs

Because algorithms are described in TLA rather differently from what you may be used to in programming, this would be a good point to ask, what is an algorithm?

In that 1979 letter to the editor of CACM, Lamport explains how he sees the difference between an algorithm and a program:

> *The word "algorithm" usually means a general method for computing something, and "program" means code that can be executed on a computer.*

Bob Harper, a programming language theorist, disagrees and **writes that**:

> *Algorithms are programs. The supposed distinction only arose because of ultra-crappy programming languages.*

Let's examine Harper's assertion. Suppose we read on Wikipedia a description of Tony Hoare's famous Quicksort algorithm — which we will formally specify later in this post — and implement it in, say, BASIC and Pascal. Do the two programs encode the same algorithm or not? If we say they do, then Harper's statement is taken to mean "algorithms are programs modulo some equivalence," but this requires us to define what it means for programs to be equivalent, or "essentially the same", and it is then that equivalence — which is bound to be far from trivial — that captures the essence of what an algorithm is rather than a specific program. Indeed, **a paper** I will discuss briefly in part 4, defines an algorithm precisely in this way, an equivalence relation on programs, and shows that it is the resulting "quotient" that captures the more important essence of what we mean when we say algorithm. On the other hand, if we do *not* consider the two programs to be the same algorithm — perhaps because the two languages have some differences in implementation, concerning, say, how arrays are represented in memory, and we consider those differences significant — then the very same Lisp program could actually encode several different algorithms depending on what version of compiler or interpreter we use to run it, as different versions may have implementation details that are just as different as those between Pascal and BASIC[7].

As an example, consider **Euclid's algorithm** for computing the greatest common divisor of two natural numbers. I picked this example because the algorithm is one of the oldest known that is still in use (according to Wikipedia, it was published in 300BC), it is very simple, and because it is a favorite example of Leslie Lamport's in his TLA$^+$ tutorials and talks. This is the algorithm: if the two numbers are equal, then they're equal to their GCD; otherwise, we instead consider the pair comprising the smaller of the two numbers and the difference between the two numbers, and repeat.

Now, here's a simple question that's a favorite in some online programming forums: is Euclid's algorithm imperative or pure-functional? We cannot answer this question because the algorithm doesn't talk about specific memory operations, doesn't say anything about the scope and mutation rules for variables, and doesn't dictate or forbid the use function abstractions when programming it. Nevertheless, it is no doubt an actual algorithm. We can even reason about it and prove its correctness — so we understand exactly what it means and why it works — yet it leaves unspecified this particular detail, which turns out to be irrelevant to understanding how and why the algorithm works. The lack of this detail is not a flaw, let alone one due to Euclid's choice of an ultra-crappy language to describe his algorithm (although, personally, Greek would not be my first choice).

An algorithm provides some details and leaves others unspecified; we say that an algorithm works if it does what it claims to do no matter how we fill in the gaps in the description[8]. A program, being a description of an algorithm in a (formal) programming language, is a description at a specific level of detail fully dictated by the language in which it is expressed. It can provide neither more nor less detail than that required by the language. A Python program cannot define precisely how its data is to be laid-out in memory, and when writing a mergesort algorithm, it cannot leave out the detail of whether it is to be done sequentially or in parallel (some languages may give more leeway, but even then they have some fixed range of specificity). When Euclid's algorithm is implemented in some programming language, we usually have no trouble determining whether the algorithm is imperative or pure-functional, yet, regardless of the answer to that, we also have no trouble determining whether or not the program actually implements Euclid's algorithm. Qualities that seem so essential to discussions and endless debates about programming, are irrelevant detail for the algorithms our programs are ultimately written to carry out.

To drive this point home, let's consider Quicksort again. This is the algorithm **according to Wikipedia** :

> 1. *Pick an element, called a pivot, from the array.*
> 2. *Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.*
> 3. *Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.*

I claim that *none* of the steps of this algorithm is directly expressible as a program in virtually any well-known programming language. The first step requires picking an element in the array; but which element is it and how is it picked? There are many different choices. The second step requires reordering the elements of the array, but there are many different reorderings that satisfy the partitioning requirement. The third step calls for recursively applying the steps to two sub-arrays, but doesn't mention in which order, or whether in any sequential order at all; maybe we can work on the two sub-arrays in parallel.

Without those details, we simply cannot write a program that implements Quicksort. However, those details are not missing from the algorithm, which is complete as given; they simply do not matter, or, in other words, their particulars are not assumed. The algorithm works *no matter* which element we pick as a pivot; it works *no matter* which particular reordering we choose as long as its a proper partition; it works *no matter* in what order we execute the recursive steps or even if we run them in parallel. In fact, any description that does fill in those details — and

therefore any Quicksort *program* — cannot possibly fully describe Quicksort, but merely one of its many possible implementations.

If we could describe the general Quicksort algorithm, we can verify that it is correct (i.e. that it sorts its input). Once we've done that, we no longer need to prove that a particular implementation of the algorithms is correct — we just need to prove that it is indeed an implementation of the algorithm. TLA+ allows precisely this form of verification, namely, verifying that some property holds for a high-level specification, and then verifying that a low-level specification implements the high-level one.

Now, you may think that as a programmer, rather than a researcher who publishes algorithms in scientific journals, you do not care about the distinction between an algorithm and a program because you're only interested in programs, anyway. But you would be wrong. You are probably required to design large software systems, not just code isolated subroutines, and a system resembles an algorithm much more than a program. When you design the system, you don't care about the details of which text search product will be used, which message queue will be picked, which database will store your data, and which MVC framework will be used on the client. You may very well be interested in *some* of their properties — for example, whether or not the message queue can drop messages, the consistency guarantees of the database, the performance of the search, etc. — but there are many more details you don't care about. Nevertheless, you still need to know that the system will meet its requirements *no matter* which component is chosen (as long as it satisfies the requirements you do care about). You therefore need to test the correctness of a design that cannot be detailed enough to actually run, and it must not be: if it is too detailed, its behavior may accidentally depend on some database quirk that all of a sudden goes away in the next release. Even if your components are all picked in advance and you're sure that they won't change, as I said in part 1 verifying a specification that is very detailed, spanning everything from the code-level to high-level architecture is currently infeasible in practice. The ability to specify *and verify* a high-level design without supplying low-level details is necessary for a formal method to scale.

TLA gives a quite satisfying answer to the question of what an algorithm is (and how algorithms of different levels of detail relate to one another): an algorithm is quite literally the collection of all its implementations. Another way to look at it is that an algorithm is the common element among all of its implementation; yet another is that an algorithm is an abstraction — a *less* detailed description — of its implementations and a refinement — a *more* detailed description — of algorithms it implements itself, namely its own abstractions. This yields a **partial-order relation** on all algorithms, called the abstraction/refinement relation.

Some programming languages that include specification mechanisms (like Eiffel, SPARK and Clojure) also allow you to specify what an algorithm does in some language-specific level of detail, and those specifications also denote the set of all possible implementations, but they still make a very clear distinction between a specification and a program, the latter being the only thing that's executable. The same is true in sophisticated research programming languages (like Agda and Idris) that make use of **dependent types** as a specification language. Even though the type definitions (like specification contracts) allow full use of the language's *syntax*, there is a clear-cut, binary distinction between a specification — a type — which can be at some relatively flexible level of detail, and a program or a subroutine — a type inhabitant — whose level of detail is predetermined by the language's particular semantics. Types and their inhabitants, even in dependently-typed languages, are two distinct semantic categories. You cannot execute a type, nor can a subroutine directly serve as a specification for another subroutine. TLA makes no such distinction; one algorithm can implement another, and "running" an algorithm means examining all of its possible behaviors, whether there are very few of them (as in a program) or very many (as in a non-detailed algorithm).

An algorithm is, then, something more general than a program. Every program is an algorithm, but not every algorithm is a program. Nevertheless, there is no *theoretical* distinction between a program and an algorithm; they differ in measure not in quality, and what constitutes sufficient detail to qualify as a program depends on the choice of a programming language. In fact, just as an algorithm can implement another algorithm by adding more detail, and a program can implement an algorithm, so too can a program implement another program, although that

usually requires both programs to be written in different languages. For example, two different Lisp compilers produce two different machine code programs, both implementing the same Lisp program. I will therefore be using the two terms, algorithm and program, somewhat interchangeably unless it is clear from context that only one applies.

One of the nice things TLA+ teaches you is to think of an algorithm as an abstract concept. There is no computer (unless you want to simulate it), and no compilation — just a mathematical description, not unlike a mathematical model of a physical system. This lets you carefully think about what the algorithm is all about, or what it really does, but you can choose at what level of detail you want to describe it, what level of detail captures the essence of the algorithm. If you wish, you can say, "this algorithm sorts" adding no additional information; if you wish you can describe what's required to happen all the way down to logic gates in the processor and their power consumption. This provides us with a nice separation of concerns: you can think about how an algorithm or a system works separately from how to cleanly organize the code that implements it.

Time in Computation

One of my goals in this series is to compare the mathematical theory of TLA+ with others mathematical theories of computation and programming. A much talked-about mathematical theory of programming, especially among programming-language researchers and enthusiasts, is the theory of functional programming. At its core, functional programming of the pure variety chooses to ignore the notion of time in computation, rather representing a program as a composition of functions. This is a major contrast with TLA, which represents the passage of (logical) time very explicitly. It is a direct result of this representation of time that programs with side effects, pure computation, batch programs, interactive programs, sequential programs, concurrent programs, parallel programs, distributed programs — any kind of program imaginable — are represented in exactly the same way, and reasoning about all kinds of programs is also done in the same way. In fact, it is impossible to formally tell the difference between a program that does pure calculations and one that has lots of side-effects in TLA[9]. In fact, it is also impossible to formally tell the difference between sequential, concurrent and parallel programs in TLA. The difference between them lies entirely in how we choose *interpret* TLA formulas.

I believe that the first thorough, formal treatment of time in the context of programming was the **situation calculus**, introduced by John McCarthy (Lisp's inventor) in his 1963 paper, ***Situations, Actions, and Causal Laws***[10], as part of his research of AI. He writes:

> *Intuitively, a situation is the complete state of affairs at some instant of time. The laws of motion of a system determine from a situation all future situations. Thus a situation corresponds to the notion in physics of a point in phase space. In physics, laws are expressed in the form of differential equations which give the complete motion of the point in phase space.*

Around the same time, Arthur Prior was working on **temporal logic**, a simpler formalism than McCarthy's which was later introduced into computer science by Amir Pnueli, in his paper ***The Temporal Logic of Programs***. Pnueli describes his formalism as one "in which the time dependence of events is the basic concept". TLA is simpler still.

While I pointed out in part 1 that preference for different formalisms is often a matter of aesthetic taste, the fact that all kinds of computation are expressed in TLA in one simple manner — because of the treatment of time — that questions of side-effects or concurrency not only do not pose any serious difficulty but are entirely a non-issue, and that reasoning about different kinds of programs can be done in exactly the same straightforward way, at least suggests that TLA is a formalism that describes computation in a very "natural" way, meaning there is little friction between the formalism and what it seeks to model. This harmony, as we'll see, comes at no cost in terms of our ability to talk about programs in very abstract or very concrete ways.

Introduction to TLA

TLA describes algorithms not as programs in some programming language but as mathematical objects, detached from any specific syntactic representation, just like the number four represents the same mathematical object whether it is written as the arabic numeral 4 or as the roman numeral IV[11]. TLA views algorithms and computations as discrete dynamical systems, and forms a general logical framework for reasoning about such discrete systems that is similar in some ways to how ordinary differential equations are used to model continuous dynamical systems, but with constructs that are of particular interest to computer scientists.

In ***The Temporal Logic of Actions***, Lamport writes:

> *A[n]… algorithm is usually specified with a program. Correctness of the algorithm means that the program satisfies a desired property. We propose a simpler approach in which both the algorithm and the property are specified by formulas in a single logic. Correctness of the algorithm means that the formula specifying the algorithm implies the formula specifying the property, where implies is ordinary logical implication.*
>
> *We are motivated not by an abstract ideal of elegance, but by the practical problem of reasoning about real algorithms. Rigorous reasoning is the only way to avoid subtle errors… and we want to make reasoning as simple as possible by making the underlying formalism simple.*

TLA itself does not specify how data and primitive operations on it are defined, but only concerns itself with describing the dynamic behavior of a program, assuming we have some way of defining data and data operations. TLA therefore requires a "data language" or a "data logic". In part 2 we saw the data logic provided by TLA+.

Let's now look at how Euclid's algorithm is expressed in TLA+. Note that we could use Euclid's algorithm to *define* a GCD operator as we learned in part 2, like so:

$$
\begin{aligned}
GCD(m, n) \triangleq \ & \text{CASE} \ \ m = n \to m \\
& \ \ \square \ m > n \to GCD(m - n, n) \\
& \ \ \square \ m < n \to GCD(m, n - m)
\end{aligned}
$$

But this does not describe an algorithm in TLA+; it is just another mathematical definition of the greatest common divisor, and not a very good one, as it is unclear that the greatest common divisor is the object being defined. Rather, this is Euclid's algorithm in TLA+:

$$
\begin{aligned}
& \text{CONSTANTS} \ M, N \\
& \text{VARIABLES} \ x, y \\
\\
& Init \triangleq x = M \land y = N \\
& Next \triangleq \ \lor \ x > y \land x' = x - y \ \land y' = y \\
& \qquad \lor \ x < y \land x' = x \qquad \land y' = y - x
\end{aligned}
$$

assuming that we treat either x or y as the "output" once the algorithm terminates (there are some missing details here, like how $Init$ and $Next$ are combined to form a single formula, as well an important detail of how termination is modeled, but this is "morally" correct, and, in fact, all you need to try the algorithm in the TLC model checker).

Your first objection may be that unlike my original English description of the algorithm, in this one you *can* tell that the algorithm is imperative because it contains the expression $x' = x - y$, which looks like an imperative

assignment. It is not. All it means is that the variable, i.e. the *name*, x will refer to the value $x - y$ in the next program step. If you look at a pure functional implementation of the algorithm, you see that the same happens there:

```
gcd(x:Int, y:int) =
        if x = y then x else
        if x > y then gcd(x-y, y) else
        if x < y then gcd(x, y-x)
```

The variables `x` and `y` stand for different values at different times in the program's execution, yet no imperative assignment is involved.

Another objection may be that, like a program, the above specification is only one particular description of the algorithm, where others could be chosen. For example, instead of two separate variables, x and y, I could have used a single variable assigned a pair of values. However, that seemingly different description would actually be equivalent to the one above; it would still be the same algorithm in a very precise sense. TLA defines an equivalence relation under which two formulas that describe the same algorithm *informally* also describe the same algorithm formally.

A *property* of an algorithm can be something like "returns a positive, even integer", or "runs in $O(n \log n)$", or "finds the greatest common divisor". Properties are things that, in the programming languages I mentioned above, you'd normally describe in a contract (Eiffel, SPARK, Clojure) or as a type (Agda, Idris). But in TLA, just as one algorithm can specify another by being a more abstract, less detailed, description — or, in other words, an algorithm can be a "type" — so too can a type describe an algorithm; as we'll see, there is no distinction in TLA between an algorithm and an algorithm property.

TLA describes sequential, concurrent, parallel and even quantum algorithms — and their properties — all using a simple formalism on top of the logic that specifies data and operations/relations on it (like the one we've seen in part 2), of which only equality, $=$, is strictly required by TLA. It has just four constructs — $'$, $\Box$, $\mathrm{ENABLED}$, $\exists$ — only one of which, prime, is absolutely necessary in practice if all you want to do is specify programs and check their correctness in the TLC model-checker. This minimalism is a testament to the power and elegance of the logic.

The Standard Model, Safety and Liveness

We will describe TLA in two iterations: first a simple but naive version of TLA which Lamport calls "raw" TLA, or rTLA, and then, after describing the problems with rTLA, we'll get to TLA, which fixes those problems.

TLA (and rTLA), is a logic, and a logic has both syntax and semantics. I will start with the semantics or, rather, with the conceptual framework that will form the semantics of the logic. This is a natural place to start because this was also the chronological order of development of the ideas behind TLA.

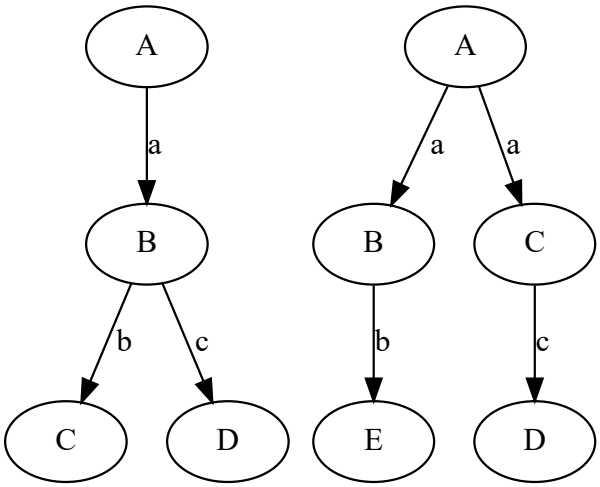
Computation As a Discrete Dynamical System

There are many ways to describe computation. One way is to describe a computation as a mathematical function. But this description is too imprecise. For example, a bubble-sort algorithm and a merge-sort algorithm both compute the same function (from a list of, say, integers, to a sorted list of the same integers), but if our formalism equated computation with functions, we would not be able to tell the two apart, and we know that the two have different properties — like their computational complexity — that we may be interested in, and that a universal formalism must be able to reason about. In addition, modeling important classes of algorithms, like interactive and concurrent systems, as functions is inconvenient, requiring **a different treatment** from sequential computations. For example, while even the most complex of compilers can be described as a single function from input to output, a trivial clock program that continuously displays the time cannot.

Instead, we will describe a computation as a discrete dynamical process. A dynamical process is some quantity that is a function of *time*. If time is continuous, we say that the dynamical process is continuous; if time is discrete (and so most likely denotes some logical time rather than physical time), the process is discrete. A continuous process can be expressed mathematically as a function of a positive real-valued time parameter. A discrete process is a function of a positive integer parameter or, put simply, a sequence, usually called a behavior or a trace.

But a sequence of what exactly? We have a few choices. We can have a sequence of what the program *does* at each step; that would make it a sequence of *events* (usually, the word *action* is used instead of event, but I will use event so as not to confuse it with the different concept of action in TLA). An event can be thought of as some output that the program emits or an input it consumes; it can be thought of as an *effect*. Another alternative is to describe a computation as a sequence of program states. Yet another is as a sequence of states *and* events (each event occuring at the transition between two consecutive states).

Process calculi (like the **π-calculus** or **CSP**) use a trace of events to describe a computation. This choice was made because process calculi use the language-based, or algebraic approach to specification, and programming languages don't like to explicitly mention the program's state, which includes control information like the program counter (the index of which instruction is to be executed next). But this choice has a major drawback on reasoning. Consider the following two flowcharts describing two programs (branching indicates nondeterministic choice):



Both programs generate the same two possible traces: $\{a \to b, a \to c\}$, so they are similar in some sense, yet they are clearly different in another. This means that an algorithm cannot be uniquely defined by its set of possible traces, and we will therefore need some other — more complicated — notion of equivalence[12]. After all, a definition of any kind of mathematical object must include a notion of equivalence that defines when two objects of that kind are the same.

This leaves us with representing a computation as a sequence of states or a sequence of states *and* events. We note that a sequence of states and events $s_1 \overset{e_1}{\longrightarrow} s_2 \overset{e_2}{\longrightarrow} s_3 \ldots$ can be easily represented as this sequence of states alone: $\langle s_1, - \rangle \to \langle s_2, e_1 \rangle \to \langle s_3, e_2 \rangle \ldots$ So we'll pick the simpler option, use a sequence of states. This is also nicely analogous to continuous systems, which are also functions from time to state.

The Standard Model

We define a computation — an execution of an algorithm — as a *behavior*. A behavior is an infinite sequence of *states*, and a state is an assignment, or mapping, of *variables* to *values*. What values a variable can take is defined separately by what we'll call the *data logic*; in part 2 we've seen that in the data logic of TLA+, a value is any set of ZFC set theory. While behaviors are *always* infinite, we call behaviors that have a terminal state that is reached in some prefix of the infinite sequence and then never changes *terminating* behaviors. An *abstract system* is a collection of behaviors. This is what Lamport calls *the standard model*.

In this post, will use the term "set" loosely, and interchangeably with "collection", but note that a collection of behaviors may or may not be a formal set in the set theory sense, depending on the data logic. For example, in TLA$^+$, since a variable can take any set as a value, a collection of behaviors may be "too large to be a set" in ZFC, meaning it may not be a set that can be constructed using the ZFC axioms, but rather a **proper class**.

An *algorithm* is a kind of an abstract system because we can think of it as something that generates different behaviors, different executions. Note that not every abstract system, i.e., a set of behaviors, can be reasonably called an algorithm because we require of algorithms that they have some finite description, and by a simple cardinality argument, there are many more sets of behaviors than there are possible strings of finite length. In addition, there may be other limitations (such as computability) that depend on what operations our data logic allows. But this is only a starting point. We will gradually refine the notion of an algorithm so that it nicely coincides with out intuitions as well as other definitions.

How does an algoritm generate more than one behavior? Because an algorithm allows for *nondeterminism*, which simply means that the specification allows for more than one state at some step of the program. What is the source of the nondeterminism? Input from the environment is one such source. Notice that we didn't define the algorithm in any parameterized way, like "a function of an integer argument". If an algorithm is a deterministic subroutine with a single integer parameter called, say, x, then in our standard model the algorithm will be the set of all of the routine's behaviors, one for each possible integer value of x. Another source of nondeterminism may be the behavior of the operating system as it schedules processes. At any step, the OS may execute an instruction of one thread of the application or of another. However, the most important source of nondeterminism is our own choice of level of detail. We may choose to specify an algorithm down to the level of machine instructions or we may leave some details unspecified. The most important insight is that all of these kinds of nondeterminism are actually of all of the same kind — the last. When we specify *anything*, we describe only certain aspects of its behavior; nondeterminism is all the rest.

A *property* of a behavior is any collection of behaviors. For example, "the x variable is a natural number between 3 and 5" is the set of behaviors whose x variable is always between 3 and 5.

Now, what is a property of an *algorithm* or a system? If a property of behaviors is a set of behaviors, then a property of systems should be a set of systems, namely a set of sets of behaviors. However — and this is a crucial point in the design and theory of TLA — if we are willing to restrict the kind of algorithm properties we allow, we can simplify things greatly. If we only allow discussing properties that are true of an algorithm if and only if they are true of *all* its behaviors — those that say what an algorithm must or must not do in *every* execution — then we can define a property of an algorithm to also be just a set of behaviors. The algorithm satisfies a property iff it — i.e. the set of its behaviors — is a subset of the property. There is no ambiguity.

How can we enforce this restriction? By making our logic *first-order*. As we saw in part 2, in a first-order logic, the model of any formula is a set of objects in the logic's structure. What does this buy us? This means that properties of behaviors, algorithms, and properties of algorithms are all the same kind of objects — a collection of behaviors. This is an extremely powerful idea. One of the selling points of research programming languages with dependent types is that the algorithm properties they specify — their types — can make full use of the language syntax. Semantically, however, types and programs are completely distinct: you can't run a type and you can't use a program as a property (at least, not in the same sense as in TLA). This separation makes sense for a programming language, as it ensures that programs are specified at a level that's fit for efficient execution, but unifying programs and programs properties makes a formalism for reasoning simpler.

Safety and Liveness

Let us now define two kinds of properties: safety and liveness. We don't need to say whether we're talking about a property of behaviors or a property of algorithms, as the two are the same in our framework. A safety property,

intuitively speaking, specifies what a behavior must *never* do. For example, "x is always an even number", is a safety property that is violated if x is ever not an even number. A liveness property says what a behavior must *eventually* do. For example, "x will eventually be greater than 100". Where sequential algorithms are concerned, *partial correctness*, meaning the assertion that *if* the program terminates then the result it produces will be the expected one, is a safety property. The only interesting liveness property of a sequential algorithm is termination, meaning the assertion that the program will eventually terminate. Interactive and concurrent programs, however, may have many interesting liveness properties, such as, "every request will eventually yield a response", or "every runnable process will eventually run". Worst-case computational complexity, of time or space, is a safety property, as it states that the algorithm must never consume more than some specific amount of time or memory. It may seem obvious that the safety property of worst-case time complexity implies the liveness property of termination, but actually, whether it does or not depends on the specific mathematical framework used (to see how it may not, consider that there may be different relationships between a state in a behavior and what constitutes a program step in the context of time complexity, or different ways to map the states in the behavior to time; but we'll get to all that later).

The names "safety" and "liveness" in the context of software verification were coined by Lamport in his 1977 paper ***Proving the Correctness of Multiprocess Programs***, and their interesting topological properties, which we will now explore, were discovered by Bowen Alpern and Fred Schneider in their 1987 paper, ***Recognizing Safety and Liveness***. This part is among the least important part of the theory of TLA+ for practical purposes, but I personally find the view of computation from the perspective we'll now explore to be extremely interesting.

Notice the following characteristics of safety and liveness properties: because a safety property says that nothing "bad" must *ever* happen, a behavior violates it iff some finite prefix of it (remember, behaviors are always infinite) violates the property, as the "bad" thing must happen at some finite point in time in order for the property to be violated. In contrast, a liveness property — something "good" must *eventually* happen — has the quality that any finite prefix of a behavior could be extended to comply with the property by adding appropriate states that eventually fulfill the property.

If σ is some behavior, we will let σ^n denote the prefix of length n of the behavior. If we have an infinite sequence of behaviors, $\sigma_1, \sigma_2, \ldots$ we'll say that the behavior σ is their limit, namely $\sigma = \lim_{i \to \infty} \sigma_i$ or just $\sigma = \lim \sigma_i$, iff, intuitively, the prefixes of the behaviors *converge* to σ. More precisely, $\sigma = \lim \sigma_i$ iff for every $n \geq 0$ there exits an $m \geq 0$ such that $i > m \Rightarrow \sigma_i^n = \sigma^n$. If S is a set of behaviors, we say that the behavior σ is a *limit point* of S, iff there are elements $\sigma_1, \sigma_2, \ldots$ in S such that $\sigma = \lim \sigma_i$.

This definition of a limit allows us to turn our universe of behaviors into a **topological space**[13], by defining a set S of behaviors to be *closed* iff it contains all of its limit points. We then define the *closure* of any set S of behaviors, denoted $\overline{S}$, as the set of all limit points of S. $\overline{S}$ is the smallest closed superset of S, and S is closed iff $S = \bar{S}$. We will say that a set S of behaviors is *dense* iff any behavior whatsoever in the space is a limit point of S.

Back to safety and liveness: Let P be some safety property (i.e. a set of behaviors). If a behavior σ does not satisfy P, i.e. $\sigma \notin P$, then property is violated by some finite prefix of the behavior. Let's say that the first state that violates the behavior is at index n. By the definition of convergence, if some subset $\sigma_i \subseteq P$ were to converge to σ then all elements of the sequence of σ_i would need to coincide with σ^n starting at some index, but that would mean that all of the elements after that index would no longer be in P and we get a contradiction. This means that any $\sigma \notin P$ cannot be a limit point of P, which means that P contains all its limit points; a safety property is therefore a closed set. In fact, this works in the other direction, too: every closed set can be distinguished by a finite prefix, and so every closed set is a safety property. Now, let L be some liveness property, and let σ be any behavior whatsoever. As we said, any finite prefix of any behavior could be extended by adding more states so to it so that it is in L. This means that for any finite prefix σ_i of σ, we can create a behavior $\hat{\sigma}_i$ such that $\hat{\sigma}_i \in L$ but a prefix of length i of $\hat{\sigma}_i$ is equal to σ_i i.e. $\hat{\sigma_i}_i = \sigma_i$. This means that $\sigma = \lim \hat{\sigma}_i$. But as σ is completely arbitrary, this means that every behavior in the entire space is a limit point of L; a liveness property is a dense set. This works in the other direction,

too: if L is some dense set, and S is any set, every finite prefix of a behavior in S could be extended by adding states so that it lies in L; a dense set is a liveness property.

It is a theorem that every set in a topological space is equal to the intersection of some closed set and some dense set. This means that any system, and therefore any algorithm, is an intersection of a safety property, stating what "bad" things must never happen, and a liveness property, stating what "good" thing must eventually happen.

Temporal Formulas

Let us now turn our attention to the syntax of the logic. TLA (and its simplified version, rTLA, which we're starting from) is a **temporal logic**, which is a kind of **modal logic**. Whereas in the ordinary, non-modal logic we saw in part 2, a model is an assignment of values to variables that makes the formula true, in modal logic there are multiple *modalities*, or worlds, in which variables can take different values; $x = 1$ in one world and $x = 2$ in another. A variable that can have different values in different modalities is called a *flexible variable*. A variable that must take the same value in all modalities is called a *rigid* variable. In temporal logic, the different modalities represent different points in time. $x = 1$ at one point in time and $x = 2$ in another.

There are two basic kinds of temporal logic. The first, called **linear temporal logic**, or LTL, views time as linear, and the time modalities are points on a line. The second, **computation tree logic**, or CTL, views time as branching towards many possible future, and the time modalities are nodes of a tree. Interestingly, LTL and CTL are *incomparable*, meaning there are things that can be expressed in one and not the other[14]. There is, however, a consensus that LTL is easier to use, and probably more useful[15], TLA is a linear-time logic (although it only borrows one construct from LTL), as that fits with our view of computations as behaviors, which are sequences rather than trees.

I will call the flexible variables of our logic *temporal variables*, and like the rigid variables — declared with the $\mathrm{CONSTANT}$ keyword or introduced with the quantifiers $\forall$ and $\exists$ — temporal variables are declared with the keyword $\mathrm{VARIABLE}$ (or its synonym $\mathrm{VARIABLES}$), or introduced with a temporal quantifier. A temporal variable may have a different value at each state of a computation.

Whereas a model for a formula in an ordinary logic is an assignment of values to variables that satisfies the formula, a model for a temporal formula is an assignment of values to variables in *all* modalities, i.e., all states of a behavior. The model, or meaning, or formal semantics of a temporal formula is, then, the set of behaviors that satisfy it. We will explain how a temporal formula is constructed by building it up from different kinds of simple expressions.

Our expressions have four syntactic *levels*. An expression that does not refer to any temporal variable, either directly or through some definition it refers to, is called a *constant* expression (level-0). An expression that does refer to variables, either directly or through a definition (and doesn't also contain the operators we haven't yet discussed) is called a *state function* (level-1), because it is some function of the state (remember, the state is the value of all temporal variables). For example, if x and y are temporal variables (declared with $\mathrm{VARIABLES}$), then the expression x is a state function, as is $x * 2$, as is $x + y$. So is the expression $x < y$, but because that state function denotes a boolean value[16] we will call it a *state predicate* (which is just a name for a boolean-valued state function).

Now things get more interesting. As every expression relates to a modality — or a point in time — the "current" state is always defined, and as our temporal modalities are arranged as an infinite sequence, a "next" state is also always defined. So, if x is a temporal variable, then x' (read, "x prime") is the value of x in the *next* state. If the expression e is a state function (or a state predicate), then e' is equal to the expression e with all the temporal variables in it — and in definitions it references — primed. So the value of the state function e' is the value of the state function e in the *next* state. An expression that contains primed variables or primed subexpressions is called a *transition function*, or, if it is a predicate, a *transition predicate* (level-2). Transition predicates are better known as *actions*, the very same actions that give TLA its name and form its very core.

Only constant and state-function (i.e. level-0 and level-1) expressions can be primed. Priming an expression that already contains primed sub-expressions or other temporal operators we'll get to soon, is a syntax error, and is ill formed. Of course, as a constant has the same value in all states, priming it has no effect.

Actions, or transition predicates, are predicates not of a single state but of two states, one denoted with unprimed variables, and the other with primed variables. If the predicate is true, then the primed state is a possible next-state of the current, unprimed state.

For example, the expression $x' = 1$ is a transition predicate — an action — which states that the value of x in the next state will be 1. The expression $x' = x + 1$ is also an action, this time one that says that the value of x will be incremented by 1, as is $x' = x + y$, or $x' \in Nat \land x' < y'$, which says that the next value of x will be some natural number which is less than the next value of y.

If A is an action, $\text{ENABLED } A$ is a *state* predicate stating that there exists *some* next state that satisfies A. For example, if:

$$\begin{aligned} A \triangleq & \land x\%2 = 0 \\ & \land \lor x' = x/2 \\ & \quad \lor x' = -x/2 \end{aligned}$$

then A could be read as "if x is even in the current state, then in the next state, x could be either $\frac{x}{2}$ or $-\frac{x}{2}$". A specifies a next state only if x is currently even, so, in this case, $\text{ENABLED } A \equiv x\%2 = 0$.

For the sake of completeness, I will mention the action composition operator, $\cdot$, even though it is unsupported by any of the TLA+ tools, and its use is discouraged. If A and B are actions, then $A \cdot B$ is the action that is true for the transition $s \rightharpoonup t$ iff there exists a state u such that A is true for $s \rightharpoonup u$ and B is true for $u \rightharpoonup t$; basically, it's an A step followed by B step rolled into a single transition.

So far we've been talking about expression, but now have the basic components to start talking about formulas. If F is a formula and σ is a behavior, we will write $\sigma \models F$ if F is true for the behavior, or σ *satisfies* F. The meaning, or semantics of F (which we write as $[\![F]\!]$) is all behaviors σ such that $\sigma \models F$. I will denote — unlike in the previous section — the (i+1)'th state of σ as σ_i, so $\sigma = \sigma_0 \to \sigma_1 \to \sigma_2 \to \ldots$ Also, $\sigma^{+n} = \sigma_n \to \sigma_{n+1} \to \ldots$, namely the *suffix* of σ, with its first n states removed.

We will define well-formed formulas recursively. If F and G are formulas, then so is $F \land G$, and a behavior σ satisfies $F \land G$ iff it satisfies both F and G (formally $\sigma \models (F \land G) \equiv (\sigma \models F) \land (\sigma \models G)$). $\neg F$ is a formula, and is satisfied by a behavior σ iff σ does *not* satisfy F. Similarly, we can define the meaning of all other connectives, but let's take a look at $\Rightarrow$: $\sigma \models (F \Rightarrow G) \equiv (\sigma \models F) \Rightarrow (\sigma \models G)$. This means that if F and G are formulas, $F \Rightarrow G$ iff every behavior that satisfies F also satisfies G, or, in other words, the set of behaviors defined by F are a subset of those of G's. This relation forms the core of what it means for an algorithm to implement another or satisfy a property, and we'll take a closer look at it later.

Now, if P is a state predicate then it is also a formula, and $\sigma \models P$ iff the predicate is satisfied by the *first* state of the behavior. For example, if x is a temporal variable, the formula $x = 3$ denotes all behaviors in which x is 3 in the first state.

If A is an action — namely a transition predicate, or a predicate of *two* states — then it can also serve as a formula, and $\sigma \models A$ iff the action is satisfied by the *first two* states of the behavior. For example, if x is a temporal variable, the formula $x = 3 \land x' = x + 1$ denotes all behaviors in which x is 3 in the first state and 4 in the second. The formula $x \in Nat \land x' = x + 1$ denotes all behaviors in which x is some natural number in the first state and is incremented by 1 in the second.

Now lets introduce the temporal operator $\Box$, borrowed from LTL. If F is a formula then $\Box F$ is also a formula, and $\sigma \models \Box F$ iff $\sigma^{+n} \models F$ for all n (i.e. $\sigma \models \Box F \equiv \forall\, n \in Nat : \sigma^{+n} \models F$).

Therefore, if P is a state predicate, then $\Box P$ means that P is true for *every* state, because $\sigma \models \Box P$ iff $\sigma^{+n} \models P$ for all n and the first state of σ^{+n} is σ_n. If A is an action, then $\Box A$ is a formula that means that A is true for every pair of consecutive states, $\langle \sigma_i, \sigma_{i+1} \rangle$, because $\sigma \models \Box A \equiv \forall\, n \in Nat : \sigma^{+n} \models A$, and the first two states of σ^{+n} are $\langle \sigma_n, \sigma_{n+1} \rangle$.

How the $\Box$ operator works on general formulas, and how to quickly understand temporal formulas, will become clear when we analyze this example, taken from *Specifying Systems*:

$$
\begin{aligned}
&\sigma \models \Box((x = 1) \Rightarrow \Box(y > 0)) \\
&\quad \equiv \forall\, n \in Nat : (\sigma^{+n} \models (x = 1) \Rightarrow \Box(y > 0)) && \text{By the meaning of } \Box \\
&\quad \equiv \forall\, n \in Nat : (\sigma^{+n} \models (x = 1)) \Rightarrow (\sigma^{+n} \models \Box(y > 0)) && \text{By the meaning of } \Rightarrow \\
&\quad \equiv \forall\, n \in Nat : (\sigma^{+n} \models (x = 1)) \Rightarrow && \text{By the meaning of } \Box \\
&\qquad\qquad (\forall\, m \in Nat : (\sigma^{+n})^{+m} \models (y > 0))
\end{aligned}
$$

So a behavior satisfies $\Box((x = 1) \Rightarrow \Box(y > 0))$ iff if $x = 1$ at some state n, then $y > 0$ at state $m + n$ for all $m \in Nat$. The box operator therefore means "always" or "henceforth", and the above formula means that if x is *ever* one, then $y > 0$ *from then on*.

If F is some formula, we define the dual temporal operator $\Diamond$ (diamond) as: $\Diamond F \equiv \neg\Box\neg F$. $\Diamond F$ therefore means "not always *not* F", or, more simply, *eventually* F.

The temporal operators can be combined in interesting ways, so, for example, $\Diamond\Box F$, or "eventually always *F*", means that in all behaviors that satisfy this formula, F will eventually be true forever. As termination is defined as a state that stutters forever, such a formula can specify termination, e.g., for a formula with some tuple of variables v, the proposition $\exists\, t : F \Rightarrow \Diamond\Box(v = t)$ states that the behaviors of F terminate[17]. $\Box\Diamond F$, or "always eventually *F*" means that at any point in time F will be true sometime in the future, or, in other words, F will be true infinitely often. Or we can define an operator (built into TLA+) that says that "F leads to G", meaning that if F is ever true, then G must become true some time after that, like so:

$$F \rightsquigarrow G \triangleq \Box(F \Rightarrow \Diamond G)$$

Because $\Box$ means "always" and $\Diamond$ means "eventually" you may — quite naturally — be tempted to think that $\Box$ defines only safety properties, while $\Diamond$ defines only liveness properties, but you would be wrong on both counts (and that's a problem which will require addressing!), as we'll see in the next section.

The following pairs of tautologies hold for the temporal operators:

$$\Box(F \wedge G) \equiv (\Box F) \wedge (\Box G) \qquad \Diamond(F \vee G) \equiv (\Diamond F) \vee (\Diamond G)$$

$$(\Box F) \vee (\Box G) \Rightarrow \Box(F \vee G) \qquad \Diamond(F \wedge G) \Rightarrow (\Diamond F) \wedge (\Diamond G)$$

$$\Box\Diamond(F \vee G) \equiv (\Box\Diamond F) \vee (\Box\Diamond G) \qquad \Diamond\Box(F \wedge G) \equiv (\Diamond\Box F) \wedge (\Diamond\Box G)$$

The operators $\Box$ and $\Diamond$ are duals, and from any temporal tautology we can obtain another by substituting

$$\Box \leftarrow \Diamond \quad \Diamond \leftarrow \Box \quad \wedge \leftarrow \vee \quad \vee \leftarrow \wedge$$

and reversing the direction of all implications.

Finally, just as in addition to declaring free ordinary, i.e. rigid, or constant variables with a CONSTANT declaration, we can also introduce bounded ordinary variables with the quantifiers $\forall$ and $\exists$, so to in addition to declaring free temporal variables with VARIABLE, we can introduce bounded, or quantified, temporal variables with temporal quantifiers.

The *existential temporal quantifier*, $\exists$, is the temporal analog of regular existential quantification. $\exists\, x : F$, where x is a temporal variable, basically says, "there exists some *temporal* assignment of x such that F"; instead of a single value for x, it asserts the existence of a value for x in each state of the behavior.

For those of you interested in the finer details of formal logic, the existential temporal quantifier behaves like an existential quantifier because it satisfies the same introduction and elimination rules:

$$\frac{F[t/v]}{\exists\, v : F}\ (\exists\text{I}) \qquad\qquad \frac{F \Rightarrow G \quad (v \text{ is not free in G})}{(\exists\, v : F) \Rightarrow G}\ (\exists\text{E})$$

TLA also has the dual *universal* temporal quantifier, $\forall\, x : F \triangleq \neg\exists\, x : \neg F$. T

he existential temporal quantifier is used to hide internal state; we will talk a lot about that in part 4. The universal temporal quantifier is hardly ever used.

As with the ordinary quantifiers, tuples of variables can also be used with a temporal quantifier, so I will usually write $\exists\, v : \ldots$ to denote the general form of the existential temporal quantifier with some tuple of variables. Unlike ordinary quantifiers, however, the TLA+ syntax does not support bounded temporal quantification (e.g. $\exists\, x \in Int : F$ is illegal).

It is also possible to define a temporal CHOOSE operator in the same vein, but Lamport writes that he did not add it to TLA+ because it is not necessary for writing specifications.

Note that the four expression levels form a hierarchy. A constant predicate (level-0) is a degenerate form of a state predicate (level-1), which is a degenerate form of an action (level-1), one that states that if the predicate holds in the current state then any next state is possible, which is a degenerate form of a temporal formula (level-3).

Also note that the operators $'$ and $\Box$ (and so $\Diamond$, too) are different from the "ordinary" logical operators we saw in part 2. If the value of the variable x is 3 at some point in time, then x' is *not* the same as $3'$ (which, 3 being constant, is equal to 3). Similarly, if the value of A is TRUE at some point in time, then $\Box A$ is *not* the same as $\Box\text{TRUE}$. This is a feature (called *referential opacity*) of modal logic, and, in fact of a larger class of logics that modal logics belong to, called **intensional logic**.

Another Take on Algorithms

Actions and $\Box$ are combined to specify algorithms in the following way. Take a look at the formula (assuming $\text{VARIABLE}\ x$),

$$x = 0 \wedge \Box(x' = x + 1)$$

This formula defines the following behavior: in the first state x is 0 (recall how state predicates are interpreted), and then it is incremented by 1 at every step. We've defined what is clearly an algorithm by specifying an initial state, $x = 0$, and a transition, or a *next state* relation, $x' = x + 1$. Using a formula of the form $Init \wedge \Box Next$, where $Init$ is the initial condition and $Next$ is an action, we've defined a state machine! Now we are ready to refine the definition of an algorithm expressed in TLA: an algorithm is a state machine of the form above. This definition is still wrong, but we'll fix it later.

We will look at state machines in much more detail and see what a powerful mechanism for defining algorithms they are, but it's crucial to point out that the expression $x' = x + 1$ is absolutely not an assignment in the imperative sense, meaning, it is not `x := x + 1`. Rather, it is a relation on states called the *next state* relation, that we can denote with $\rightarrow$, such that $s \rightarrow t$ iff the state t can follow the state s in a behavior. This relation is written in TLA as an action, which is a predicate on two states given as two sets of variables: unprimed and primed. We could have just as well written $x' - x = 1$ (note: this is *not* quite true in TLA+[18], but it is "morally" true).

To drive this point home, let me give a few more examples. We can write an action, $x' \in \{-1, 1\}$ that means that in the next state, the value of x will be either 1 or -1, but we can write an equivalent action like so: $x' \in Int \land (x^2)' = 1$. This is because, as per our definition of the prime operator, $(x^2)' = (x')^2$. We can write the action $x' = x + 1 \land y' = -y$, but we can write it equivalently as $\langle x, y \rangle' = \langle x + 1, -y \rangle$. It may be hard for people used to programming languages to grasp, but this works not because there is some kind of clever binding or destructuring or unification going on here, but merely because $\langle x, y \rangle'$ is the same as $\langle x', y' \rangle$ and $\langle x', y' \rangle = \langle x + 1, -y \rangle$ is just a predicate that is true iff $x' = x + 1 \land y' = -y$. It's all just simple math[19].

That actions are transition predicates has an important consequence. What does an imperative assignment statement like `x := x + 1` say about the next value of the variable `y`? Such an assignment statement means that only the value of `x` changes and nothing else, and so `y` remains unchanged. But what does the action $x' = x + 1$ say about the next value of y? Well, the transition predicate holds true of the state transition $[x : 1, y : 5] \rightarrow [x : 2, y : 5]$ (and not of $[x : 1, y : 5] \rightarrow [x : 3, y : 5]$), but it also holds true for $[x : 1, y : 5] \rightarrow [x : 2, y : -500]$ or $[x : 1, y : 5] \rightarrow [x : 2, y : \text{``hi ''}]$. In other words, because it says nothing about the next value of y, any value is allowed. To say that y doesn't change, we can write $x' = x + 1 \land y' = y$. We can also specify that several variables don't change by writing, $\langle x, y, z \rangle' = \langle x, y, z \rangle$, but TLA+'s UNCHANGED keyword lets us write instead $\text{UNCHANGED } y$ or $\text{UNCHANGED } \langle x, y, z \rangle$.

What if we want to specify a program that takes the initial value as an input and counts up from there? We could define our algorithm using a parameterized definition like so:

$$Spec(start) \triangleq x = start \land \Box(x' = x + 1)$$

But it is better and more useful to model inputs from the environment as unknowns, or *nondeterministic* quantities in our algorithm, and instead write:

$$x \in Int \land \Box(x' = x + 1)$$

This is still a single algorithm, but now it has many possible behaviors, one for each initial value, or input.

Now let's see how we can say things about our algorithms. We'll define the following algorithm that models an hour clock, and add another definition, that states that the hour is always a whole number between 1 and 12:

$$\begin{aligned}
&\text{VARIABLE } h \\
&HourClock \triangleq h \in 1..12 \land \Box(h' = \text{IF } h \neq 12 \text{ THEN } h + 1 \text{ ELSE } 1) \\
&WithinBounds \triangleq \Box(h \in 1..12)
\end{aligned}$$

We can now make the following claim:

$$\text{THEOREM } HourClock \Rightarrow WithinBounds$$

That $HourClock$ implies $WithinBounds$ means that $HourClock$ algorithm satisfies the $WithinBounds$ property, but you can also think of it in terms of behaviors. $HourClock$ is the collection of all behaviors where h counts the hour, and $WithinBounds$ is the collection of all behaviors where h takes *any* value between 1 and 12 at each step. The behaviors of $HourClock$ are all behaviors of the property, or $[\![HourClock]\!] \subseteq [\![WithinBounds]\!]$.

But remember that there is no real distinction between a property of an algorithm and an algorithm; both are just collections of behaviors. To help you see that more clearly, we'll define the same property differently:

$$CrazyClock \triangleq h \in 1..12 \land \Box(h' \in 1..12)$$

$CrazyClock$ and $WithinBounds$ are equivalent, i.e., $WithinBounds \equiv CrazyClock$ — they specify the exact same set of behaviors — but the way $CrazyClock$ is defined makes it easier for us to think of it in terms of a (nondeterministic) algorithm: it starts by nondeterministically picking a starting value between 1 and 12, and then, at every step, picks a new value between 1 and 12.

It is therefore true that $HourClock \Rightarrow CrazyClock$, but both of them are algorithms! We say that $HourClock$ *implements*, or is *an instance of* $CrazyClock$ and that $CrazyClock$ is a *specification* of $HourClock$, or that $HourClock$ *refines* $CrazyClock$ and that $CrazyClock$ is an *abstraction* of $HourClock$. But it all means the same: all the behaviors of $HourClock$ are also behaviors of $CrazyClock$.

Logical implication in TLA corresponds with the notion of implementation. In part 4 we'll explore this general notion in far greater detail, and see that implication can denote very sophisticated forms of implementation by manipulating the subformula on the right-hand side of the implication connective.

Two Serious Problems

The logic we've defined, rTLA, suffers from two problems. Suppose we want to write an algorithm for a clock that shows both the hour and the minute:

$$\begin{array}{l}
\text{VARIABLES } h, m \\
Clock \triangleq h \in 1..12 \land m \in 0..59 \\
\qquad\qquad \land \Box(\land m' = (m+1)\%60 \\
\qquad\qquad\qquad \land m < 59 \Rightarrow \text{UNCHANGED } h \\
\qquad\qquad\qquad \land m = 59 \Rightarrow h' = \text{IF } h \neq 12 \text{ THEN } h+1 \text{ ELSE } 1)
\end{array}$$

We would very much like it to be the case that $\vdash Clock \Rightarrow HourClock$ because a clock that shows both the minute and the hour is, intuitively at least, an instance of a clock that shows the hour. Unfortunately, this isn't true in rTLA, because while the behaviors of $HourClock$ change the value of h at each step, in $Clock$'s behaviors, h changes only once every 60 steps.

There is also a problem of a slightly more philosophical nature (although it, too, has pragmatic implications). The way we've identified the notion of an algorithm with formulas has a very unsatisfying consequence. To see it, let's revisit our analogy between how continuous dynamical systems are specified with ODEs and how discrete dynamical systems are specified in TLA.

The following continuous dynamical system, $x(0) = 0 \land \dot{x} = 10$ specifies a system that begins at 0, and grows linearly with a slope of 10. It is analogous to this rTLA formula: $x = 0 \land \Box(x' = x + 10)$. We could even make this clearer by the operator $d(v) \triangleq v' - v$ and rewriting the above formula as $x = 0 \land \Box(d(x) = 10)$[20].

However, ODEs can be given *boundary* conditions rather than initial conditions. This ODE specifies the same system: $\dot{x} = 10 \land x(10) = 100$. Could the same be done in rTLA? Absolutely. We'll make time explicit ($\text{VARIABLES } x, t$):

$$
\begin{aligned}
&\land x \in Nat \land t = 0 \\
&\land \Box(\land t' = t + 1 \\
&\qquad \land x' = x + 10 \\
&\qquad \land t = 10 \Rightarrow x = 100) \quad \text{this is the ``boundary condition''}
\end{aligned}
$$

Or, if you prefer, we can use the equivalence $\Box(A \land B) \equiv \Box A \land \Box B$ to rewrite the formula as:

$$
\begin{aligned}
&\land t = 0 \land x \in Nat \\
&\land \Box(t' = t + 1 \land x' = x + 10) \\
&\land \Box(t = 10 \Rightarrow x = 100) \qquad \text{this is the ``boundary condition''}
\end{aligned}
$$

So, even though we don't tell our "algorithm" where to start — the initial condition is $x \in Nat$ — it somehow knows that it must start at 0 because that's the only way it will get to 100 at step 10.

While this may be fine for describing a behavior, it is not how we normally think of an algorithm (although this sort of nondeterminism — called "angelic"nondeterminism[21], as it helps the algorithm do the "right thing" — is frequently used in theoretical computer science to study probability classes; in fact, NP problems are precisely those that can be solved in polynomial time with the help of angelic nondeterminism). When we think of algorithms we assume that it's not a process that's allowed to use information from the future.

We could get rid of nondeterminism altogether, but nondeterminism is what gives specifications their power, as it allows us to to say exactly what we know or what we care about, and leave out things we don't know or details we don't care about. Also, there are kinds of nondeterminism that fit perfectly with our notion of algorithms, such as nondeterministic input, or nondeterministic scheduling by the OS.

Even the kind of nondeterminism we've seen only poses a problem in those formulas we'd like to consider algorithms rather than just general properties of behaviors. We would certainly like to use such nondeterminism in properties that we'd show our algorithms satisfy, i.e., we'd like to show that:

$$
\vdash \underbrace{(t = 0 \land x = 0 \land \Box(t' = t + 1 \land x' = x + 10))}_{\text{``algorithm''}} \Rightarrow \underbrace{\Box(t = 10 \Rightarrow x = 100)}_{\text{``property''}}
$$

but it would be nice to be able to tell whether a formula is a "proper" algorithm or not.

We could separate algorithms and properties into two separate semantic and possibly syntactic categories. Indeed, this is precisely the road taken by specification languages such as Coq, which use dependent types for specification: arbitrary nondeterminism is allowed only in types. This is pretty much required from programming languages that need to be able to compile programs — but not specifications of their properties — into efficient machine code, so it makes sense for the syntax rules to impose this constraint on descriptions of algorithms (namely, the body of the program).

But from a specification language designed for reasoning, not programming this would require a complicated formalism whereas we want simplicity, and it would also make the important ability of showing that one algorithm implements another much more complicated if not nearly impossible.

This problem is related to why it is not true that $\Box$ specifies only safety properties and $\Diamond$ specifies only liveness properties. In fact, $\Box$ can imply liveness properties and $\Diamond$ can imply safety properties. For example:

$$
\Box(x' = x + 1) \Rightarrow \Diamond(x > 10)
$$

and if

$$
\begin{array}{l}
\text{VARIABLES } x, t \\
RandomWalkForTenSteps \triangleq \\
\quad t = 10 \land x = 0 \land \\
\quad\quad \Box(\land t > 0 \land t' = t - 1 \land (x' = x + 1 \lor x' = x - 1) \\
\quad\quad\quad \land t = 0 \land \text{UNCHANGED } \langle t, x \rangle)
\end{array}
$$

then

$$RandomWalk \land \Diamond(x = 10) \Rightarrow \Box(x \geq 0)$$

because for x to reach 10 by the time the time t runs out, it must always increment.

Luckily, one simple solution solves all those problems.

Stuttering Invariance and Machine Closure

We'll now see how TLA fixes the two problems we encountered with rTLA. Much of the analysis in this section is taken from the paper by Martín Abadi and Leslie Lamport, ***The Existence of Refinement Mappings***, 1991, although the paper doesn't talk about any formal logic, only about abstract behaviors.

As before, we'll begin with the semantics.

Invariance Under Stuttering

For a behavior σ we'll define $\natural\sigma$ — the *stutter-free* form of σ — obtained by replacing any finite subsequence of consecutively repeating states with just a single instance of the state. For example $\natural\langle a, b, b, b, c, d, d, \ldots, d, \ldots\rangle = \langle a, b, c, d, d, \ldots, d, \ldots\rangle$. Note that the trailing repetition of an *infinite* sequence of d's is not replaced. We now define an equivalence relation on behaviors. Behaviors σ and τ are *equivalent under stuttering* (we write $\sigma \simeq \tau$) iff $\natural\sigma = \natural\tau$. Furthermore, for a set S of behaviors $\Gamma(S)$ is the set of all behaviors that are stuttering-equivalent to those of S. We say that S is *closed under stuttering*, if $\Gamma(S) = S$, i.e., if for every behavior in S all behaviors that are stuttering-equivalent to it are also in S ($\sigma \in S \land \sigma \simeq \tau \Rightarrow \tau \in S$).

Now on to syntax. We say that a formula F is *invariant under stuttering* iff its model — i.e. the collection of all behaviors that satisfy it — is closed under stuttering. A stuttering-invariant formula cannot distinguish between two stuttering-equivalent behaviors (meaning, it cannot be TRUE for one and FALSE for the other).

Because state functions/predicates only talk about one state at a time and are therefore trivially stuttering invariant, only actions can potentially break stuttering invariance. The formula $x = 1 \land \Box(x' = x + 1)$ is not invariant under stuttering because it distinguishes between $\langle 1, 2, 3, 4, \ldots\rangle$ and $\langle 1, 1, 1, 2, 3, \ldots\rangle$, as it admits the first but not the second. However, the formula $x = 1 \land \Box(x' = x + 1 \lor x' = x)$ doesn't, and is indeed stuttering invariant.

The following formula is also invariant under stuttering:

$$x = 1 \land \Box(x' = x + 1 \lor x' = x) \land \Diamond(x' = x + 1)$$

because all the added $\Diamond$ clause requires is that an increment of x occurs at least once at *some* point, namely, that x isn't *always* 1. It still cannot distinguish between behaviors that are stuttering equivalent.

If A is an action of the kind we've seen so far, and e is some state function, we'll define the actions:

$$
\begin{aligned}
[A]_e &\triangleq A \lor e' = e \\
\langle A\rangle_e &\triangleq A \land e' \neq e
\end{aligned}
$$

In practice, e is virtually always just a variable or a tuple of variables. It's common in TLA+ specifications to define a tuple of all temporal variables, $vars \triangleq \langle x, y, z \rangle$, and write $[A]_{vars}$.

Now, instead of distinct categories for algorithms and properties that complicate matters considerably, all TLA does is enforce a syntax that ensures that all TLA formulas are invariant under stuttering, and all that takes is the following syntax rules:

1. When $\Box$ is immediately followed by an action, the action must be of the form $[A]_e$
2. When $\Diamond$ is immediately followed by an action, the action must be of the form $\langle A \rangle_e$

So the formula $\Box(x' = x + 1)$ is ill-formed in TLA (but is well-formed in rTLA) — it is a syntax error. Note that $\Box(x > 0)$ is still fine, as $x > 0$ is not an action but a state predicate (there are no instances of x').

We similarly change the definition of $\exists$ to mean $\exists\, x : F$ is true for a behavior σ iff it is true for a behavior τ obtained from σ by changing the values of x and adding or removing stuttering steps.

These rules ensure the following: **every TLA formula is invariant under stuttering**, and therefore cannot distinguish between two behaviors that are stuttering-equivalent.

This solves our first problem. Let's revisit our clocks and fix their definition so that they're valid TLA by placing the actions inside $[\ldots]_e$:

$$\text{VARIABLES } h, m$$

$$HourClock \triangleq h \in 1..12 \land \Box[h' = \text{IF } h = 12 \text{ THEN } 1 \text{ ELSE } h + 1]_h \land \Box\Diamond\langle h' \neq h \rangle$$

$$\begin{aligned} Clock \quad \triangleq\ & h \in 1..12 \land m \in 0..59 \land \\ & \Box[\ \land m' = (m + 1)\%60 \\ & \quad \land m < 59 \Rightarrow \text{UNCHANGED } h \\ & \quad \land m = 59 \Rightarrow h' = \text{IF } h = 12 \text{ THEN } 1 \text{ ELSE } h + 1]_{\langle h, m \rangle} \\ & \land \Box\Diamond\langle h' \neq h \rangle_h \land \Box\Diamond\langle m' \neq m \rangle_m \end{aligned}$$

Indeed, now $Clock \Rightarrow HourClock$ (we'll later see how we can show similar facts using syntactic manipulation).

The precise significance of the conjuncts ($\Box\Diamond(\ldots)$) will be explained later (and we'll see a shortcut for writing them), but because $\Box[A]_e$ can mean that the action A is *never* taken (i.e. the initial state stutters forever), those conjuncts ensure the liveness property that the clocks keep ticking an infinite number of times. They never get stuck forever.

Stuttering invariance buys us something else, too. We can now view a TLA specification not as specifying a single system defined by a state of some given temporal variables, but as specifying all of them, and by all of them, I mean *all* discrete dynamic systems whatsoever. Thanks to stuttering invariance, a TLA formula doesn't impose any speed or pace on the algorithm or property relative to all others. A TLA formula mentions how just a small finite subset of an infinite set of temporal variables change, and essentially says "I don't know anything about any other variables"; it really means that it allows any assignment of any value to any of the variables *not* mentioned. All systems exist in the same logical universe, which allows us to compare and compose them in interesting ways. Every formula describes a tiny bit of the behavior of the entire universe. Everything else is part of the nondeterminism in the formula — anything that the formula doesn't determine.

Machine Closure and Fairness

Now, we've seen that a formula of the form $\Box[A]_e$ only specifies a safety property and cannot imply a liveness property (as it can stutter forever), but the converse — that $\Diamond F$ cannot imply a safety property — isn't true, which brings us to the second problem.

When we rewrite our rTLA formula analogous to the ODE we saw above to get a well-formed TLA formula we get:

$$\begin{aligned}&\land x \in Nat \land t = 0\\ &\land \Box[\ \land t' = t + 1\\ &\qquad\land x' = x + 10\\ &\qquad\land t = 10 \Rightarrow x = 100]_{\langle t,x\rangle}\end{aligned}$$

This seems to avoid the problem, because now, instead of a behavior that knows the future and picks the right starting value for x , we get many behaviors, some starting with $x = 0$ and hitting 100 when $t = 10$ and then continuing on incrementing x forever (or stuttering), while others begin with other values, and necessarily stutter forever at some point before the tenth step. Say we start at $x = 2$ and get $x = 102$ when $t = 10$. At that point, the predicate inside the $[]_{\langle t,x\rangle}$ will be false, but the brackets add an implicit $\lor\langle t, x\rangle' = \langle t, x\rangle$, which means that the behavior will stay at $t = 10, x = 102$ forever.

It doesn't matter even if we write our formula like so:

$$\begin{aligned}&\land t = 0 \land x \in Nat\\ &\land \Box[t' = t + 1 \land x' = x + 10]_{\langle t,x\rangle}\\ &\land \Box(t = 10 \Rightarrow x = 100)\end{aligned}$$

However, we can reclaim the power to see the future if we change the $\Box$ in the last conjunct, a safety property, into a $\Diamond$:

$$\begin{aligned}&\land t = 0 \land x \in Nat\\ &\land \Box[t' = t + 1 \land x' = x + 10]_{\langle t,x\rangle}\\ &\land \Diamond(t = 10 \land x = 100)\end{aligned}$$

We will call the initial condition and the transition relation (i.e. the $\Box[Action]$ part) — in this case, $t = 0 \land x \in Nat \land \Box[t' = t + 1 \land x' = x + 10]_{\langle t,x\rangle}$ — the *machine property* of the specification, and any conjuncts starting with $\Diamond$, the *liveness property*. It is easy to see that the machine property is a safety property. The problem is this: while $\Diamond(t = 10 \land x = 100)$ in our example is indeed a liveness property, namely every finite prefix of any behavior can be extended so that it satisfies $\Diamond(t = 10 \land x = 100)$, not every behavior *that satisfies the machine property* can be extended while still complying with the machine property. This particular liveness property interferes with the safety condition of the machine property and rules out even finite prefixes — in fact it rules out all behaviors that don't *begin* with x behaving like $\langle 0, 10, 20, 30\rangle$. It implies the safety property that x must be 0 in the initial state. When a liveness property, concerned with what will eventually happen, interferes with a safety property, concerned with what *must* hold at every state and rules out behaviors that comply with the safety property, that means that information about the future is allowed to affect the present, and that is not how algorithms should be defined.

What we'd want is to make sure that the liveness property does not interfere with the machine property, which implies that every finite prefix of any behavior *that satisfies the machine property* can be extended so that it satisfies the liveness property. If we denote the set of behaviors that comprise the machine property as M and the liveness property as L, we would like it so that L would not specify any safety property (i.e. a closed set) not already implied by M. What we want is that $\overline{M \cap L} = M$. If that condition holds, we say that the pair, $\langle M, L\rangle$ is *machine closed*, or that L is machine closed with respect to M (or simply that L is machine closed, if it is clear which

machine property we're talking about). A liveness property that is machine closed with respect to its machine property, is also called a *fairness property*.

A liveness property specifies what must eventually happen. A fairness property must do so in a way that does not dictate precisely what states are allowed at any specific time. The way this is achieved is by having the liveness property specify only that the algorithm moves forwards — makes progress — without specifying any targets it must progress towards. This can be intuitively thought about as specifying the job of the operating system and/or the CPU. Fairness can be thought of as the assumptions we make about how the OS schedules our program. Indeed, when we write a program, we operate under the assumption that it will be scheduled by an OS, but we also assume that the OS will treat our program fairly, and guarantee that it makes progress if it can. In TLA, if we care about liveness and want to prove what our program will eventually do, we must make this assumption explicit, and if our program is multi-threaded, we must state our assumptions about how multiple threads are scheduled (we will see how to model threads/processes in the next section). Both the algorithm and the scheduler can still make nondeterministic choices, but machine closure dictates that those choices can in no way be restricted by what has not yet happened, i.e., we allow nondeterminism, but not *angelic* nondeterminism. Fairness just guarantees progress.

Note that machine-closure is no longer a property of the model of an algorithm — a set of behaviors — but of how that algorithm is written as an intersection of safety and liveness properties. Indeed, two formulas can be equivalent, yet one would be machine-closed and the other won't. If we replace $x \in Nat$ with $x = 0$ in the initial condition of the specification above, we'll get one that's equivalent to it (i.e., admits the very same set of behaviors), yet is now machine-closed, as the liveness property no longer places additional restrictions on the safety property. This is to be expected, because the problem of angelic nondeterminism demonstrates that a set of behaviors is not enough to fully capture the concept of an algorithm.

But how do we know that our liveness property is a fairness property and that our specification is machine-closed? The answer is that we don't in general, but TLA+ has two built-in fairness operators, and if our liveness property is a conjunction of those operators, then we're guaranteed that it's machine-closed. The operators, WF for *weak fairness* and SF for *strong fairness* take an action A as an argument, and ensure that it will eventually occur if it is enabled "often enough". They only differ in what "often enough" means.

Weak fairness will ensure A eventually occurs if it remains enabled from some point onward; in other words, the weak fairness condition is violated if at some point $\mathrm{ENABLED}\ \langle A \rangle_e$ remains true without $\langle A \rangle_e$ ever occurring. Here is the definition of WF:

$$\begin{aligned} \mathrm{WF}_e(A) &\triangleq \Diamond\Box \mathrm{ENABLED}\ \langle A \rangle_e \Rightarrow \Box\Diamond \langle A \rangle_e \\ &\equiv \Box(\Box \mathrm{ENABLED}\ \langle A \rangle_e \Rightarrow \Diamond \langle A \rangle_e) \end{aligned}$$

Strong fairness requires $\langle A \rangle_e$ to occur infinitely often if $\mathrm{ENABLED}\ \langle A \rangle_e$ is true infinitely often, although it need not remain enabled constantly:

$$\begin{aligned} \mathrm{SF}_e(A) &\triangleq \Box\Diamond \mathrm{ENABLED}\ \langle A \rangle_e \Rightarrow \Box\Diamond \langle A \rangle_e \\ &\equiv \Box(\Diamond \mathrm{ENABLED}\ \langle A \rangle_e \Rightarrow \Diamond \langle A \rangle_e) \end{aligned}$$

Strong fairness implies weak fairness, i.e., $\mathrm{SF}_e(A) \Rightarrow \mathrm{WF}_e(A)$.

When we specify a sequential algorithm and are interested in liveness, $Fairness$ can be just $\mathrm{WF}_{vars}(Next)$, where $vars$ is a tuple of all of our temporal variables. In the next section, when we learn how threads/processes are simulated, we'll see why it may be necessary to have more than one WF/SF conjunct. Intuitively, because the $Next$ action will result in a step of one or several processes, we would need a more fine-grained specification of how the scheduler schedules the different threads, namely, we would need to state that *each* thread/process is scheduled fairly, meaning that every thread makes progress.

We can now refine yet again our definition of an algorithm, and say that an algorithm in TLA — or, at least a *realizable* algorithm — is anything that can be specified by a formula of the form

$\exists\, w : Init \land \Box[Next]_v \land Fairness$ where v is a tuple of temporal variables, and w is a tuple of a subset of those variables. Of course, each of the components can be absent (and often the temporal quantifier is absent). This is called the *normal form* of a TLA formula. Such a formula is guaranteed to be machine-closed, provided that $Fairness$ is a conjunction of a finite number of $\mathrm{WF}/\mathrm{SF}(A)$ forms, such that $A \Rightarrow Next$[22].

One can ask why we would want to allow non-machine closed specification at all. For example, in formulas that specify algorithms — namely those that contain actions rather than just state predicates — we could syntactically allow only the use of WF/SF and no other liveness property. Lamport and others answer this question in a **short note**, and say that it is often beneficial and elegant to specify an algorithm at a high-level that doesn't need to be realizable in code, where, in order to show how the algorithm works, we may want to pretend that it can see into the future. Only lower-level specifications, those that could be directly translated into program code need to be machine closed, and, of course, we may want to show how a low level, machine closed specification implements a high level, non machine closed one (general relations between algorithms will be covered in part 4).

But if we want some way to tell syntactically whether or not we're specifying realizable algorithms, at least we've now isolated our problems to liveness properties and offered a simple sufficient condition for realizability — use only WF/SF as liveness conditions (and actions that imply $Next$, but that comes naturally in most cases, as we'll see in the section about concurrency).

There is another quirk involving liveness with existential quantification, that I find interesting. Consider the following specification:

$$
\begin{array}{l}
\text{VARIABLE } m \\
Spec \triangleq \exists\, n : \land\, m = 0 \land n = 0 \\
\qquad\qquad \land\, [\text{IF } m = 0 \\
\qquad\qquad\qquad \text{THEN } n' \in Nat \qquad\qquad \land\, m' = 1 \\
\qquad\qquad\qquad \text{ELSE } \ n > 0 \land n' = n - 1 \land m' = m + 1]_{\langle m, n\rangle}
\end{array}
$$

Recall that formulas of the form $Init \land \Box[Next]_v$ are safety properties, and because they're also stuttering invariant, they do not imply any liveness properties. However, with the existential quantifier we get a specification with the single free temporal variable m, which admits all behaviors that increment m *and eventually stop*; in other words, it admits only *terminating* behaviors that increment m. Formally, $Spec \Rightarrow \exists\, k : \Diamond\Box(m = k)$. But termination is clearly a liveness property, and so this specification is not equivalent to any machine machine-closed specification involving m and no other variables. The specification $m = 0 \land \Box[m' = m + 1]_m$ (with only a machine property and no liveness) is not equivalent to our specification as it admits a behavior that increments indefinitely, while the specification $m = 0 \land \Box[m' = m + 1]_m \land \exists\, k : \Diamond\Box(m = k)$, which is equivalent to $Spec$, is not machine-closed as the liveness property rules out behaviors allowed by the machine property (with m incrementing indefinitely).

As it turns out, the cause of this phenomenon is the combination of the quantifier, which hides the n variable, and what happens at the very first step of the algorithm. That single transition from $m = 0 \to m = 1$ hides an *infinite* nondeterministic choice of n. If, instead, the nondeterministic choice were finite, say between 0 and 1000, as in:

$$
\begin{array}{l}
\text{VARIABLE } m \\
Spec \triangleq \exists\, n : \land\, m = 0 \land n = 0 \\
\qquad\qquad \land\, [\text{IF } m = 0 \\
\qquad\qquad\qquad \text{THEN } n' \in 0..1000 \qquad\qquad \land\, m' = 1 \\
\qquad\qquad\qquad \text{ELSE } \ n > 0 \land n' = n - 1 \land m' = m + 1]_{\langle m, n\rangle}
\end{array}
$$

Then $Spec$ would be equivalent to the following formula that only mentions m:

$$m = 0 \land \Box[m < 1000 \land m' = m + 1]_m$$

To see why, recall that equivalence on behaviors is stuttering-invariant, so a behavior that stutters and then resumes incrementing, e.g. $\langle 1, 2, 2, 2, 3, \ldots \rangle$, is equivalent to one that doesn't stutter, e.g. $\langle 1, 2, 3, \ldots \rangle$, unless the stuttering goes on indefinitely. So the formula allows m to be incremented as long as it's < 1000, but doesn't require it to; it may stop incrementing at any value less than 1000.

A specification that only allows hidden (i.e., of a bound temporal variable) deterministic choice from a finite set at every step is said to have *finite invisible nondeterminism*, or FIN. Like machine closure, FIN is a necessary condition for an important result regarding the existence of something called a refinement mapping, which we'll learn about in part 4 .

The Unreasonable Power of Liveness

I will end this chapter with an example of how problematic, or powerful (depending on your point of view), liveness properties can be. Consider the definition:

$$Halts(M, state) \triangleq M \Rightarrow \exists\, t : \Diamond\Box(state = t)$$

where M is an arbitrary formula, and $state$ is a tuple of variables. Note that we're using an ordinary quantifier, not a temporal one, so t is a constant. You may think that $Halts$ is suspicious because it tells you if an arbitrary algorithm halts and must therefore be undecidable, but there is nothing wrong with defining what it means to halt, and that is all the definition does; it does not claim that there exists an algorithm for deciding halting. Termination is an important property that we may wish to prove for some specific algorithms, and in order to prove it, we must be able to state it. That, in isolation, is an example of a perfectly good use of $\Diamond$.

You might fear that we can use $Halts$ to construct an *algorithm* to decide halting, as if it were a subroutine we could call, but that is not so simple. $Halts$ is a (parameterized) temporal formula, a level-3 expression, and so according to our syntax rules we can't use it in an action, as in $answer' = Halts(M, vars)$.

However, consider the specification of an "algorithm" that decides things (meaning, answers "yes" or "no"):

$$\begin{array}{l}
\text{VARIABLES } x, answer \\
\\
Decide(S, Is(_)) \triangleq \\
\quad \land\, answer = \text{``maybe ''} \land x \in S \\
\quad \land\, \Box[answer = \text{``maybe ''} \land answer' \in \{\text{``yes ''}, \text{``no ''}\} \land \text{UNCHANGED } x]_{\langle answer, x \rangle} \\
\quad \land\, \Diamond(answer \in \{\text{``yes ''}, \text{``no ''}\}) \land \Diamond(answer = \text{``yes ''}) \equiv Is(x)
\end{array}$$

Our decider simply answers for each element in the given set S either "yes" or "no". It does so simply by using the supplied operator Is which provides the decider with the answer. Seems innocuous enough, except that our decider doesn't use Is inside an action like so: $answer' = \text{IF } Is(x) \text{ THEN ``yes '' ELSE ``no ''}$, but, instead, uses it in a liveness condition that is very much not machine closed — it fully determines the action's nondeterministic choice. The final two conjuncts say this: $answer$ will eventually be "yes" or "no", and it will be "yes" iff Is is true. In the action, we just pick "yes" or "no" nondeterministically, but in the liveness property we "pull" the answer in the right direction. This *Is* to be a temporal formula, bypassing the restriction of using temporal operators in actions.

Using the decider we could define:

$$H \triangleq Decide(M, \text{LAMBDA } F : Halts(F, vars))$$

Now H "decides" — i.e., forms an "algorithm" for deciding halting. Note that we have not used any non-computable operations in the data logic. The only source of non-computability here is the use of $\Diamond$ in the decider in a way that isn't machine-closed.

Issues of fairness, however, truly matter only for concurrency experts. When writing real TLA$^+$ specifications, you'll find that you rarely use the $\Diamond$ operator (in 1000 lines of specifications, it is common to see $\Diamond$ or $\rightsquigarrow$ appear just once or twice), and when you do, it will be to specify very simple and very natural liveness properties (such as: "when we receive a request, we will eventually send a response"). If you only care about safety properties, you can ignore fairness altogether. Indeed, when we use the TLC model checker to check a formula F in normal form, if we only ask it to check for safety properties (i.e. that $\models F \Rightarrow \Box P$), it will ignore any fairness conjuncts in F as they can have no effect on safety.

It is entirely possible and reasonable to write complete specifications of large real-world systems without once worrying about liveness[23]. In fact, the entire discussion of temporal formulas and liveness appears only in section 2, "More Advanced Topics", of Lamport's book, *Specifying Systems*. The book even has a short section entitled "The Unimportance of Liveness", which I'll reproduce here:

> *[I]n practice the liveness property [of a specification]… is not as important as the safety part. The ultimate purpose of writing a specification is to avoid errors. Experience shows that most of the benefit from writing and using a specification comes from the safety part. On the other hand, the liveness property is usually easy enough to write. It typically constitutes less than five percent of a specication. So, you might as well write the liveness part. However, when looking for errors, most of your effort should be devoted to examining the safety part.*

Liveness can indeed be complicated. Lamport et al. end the note about the importance of allowing non-machine-closed specifications that I mentioned above, with these words:

> *[A]rbitrary liveness properties are "problematic". However, the problem lies in the nature of liveness, not in its definition.*
>
> *One cannot avoid complexity by definition. — Stephen Jay Gould*

We have now completed defining TLA, and explored its trickier nuances (all but one, actually, which we'll save for part 4). For those of you who may be interested in a technical discussion of this logic's expressivity, see Alexander Rabinovich, ***Expressive completeness of temporal logic of action*** (sic)[24], 1998, and Stephan Merz, ***A More Complete TLA***, 1999.

State Machines

Let us now emerge from the depths of TLA, and turn to more mundane things. In a section called "Temporal Logic Considered Confusing" of *Specifying Systems*, Lamport writes:

> *Temporal logic is quite expressive, and one can combine its operators in all sorts of ways to express a wide variety of properties. This suggests the following approach to writing a specification: express each property that the system must satisfy with a temporal formula, and then conjoin all these formulas… This approach is philosophically appealing. It has just one problem: it's practical for only the very simplest of specifications — and even for them, it seldom works well. The unbridled use of temporal logic produces formulas that are hard to understand. Conjoining several of these formulas produces a specification that is impossible to understand.*

Algorithms As State Machines

In practice, we write our TLA+ specifications as state machines, and check or prove whether they imply some simple properties, usually written as very simple temporal formulas, or whether they implement other, higher-level specifications also written as state machines (we will cover the implementation relation in part 4).

In ***Computation and State Machines*** (2008), Lamport writes:

> *State machines provide a framework for much of computer science. They can be described and manipulated with ordinary, everyday mathematics—that is, with sets, functions, and simple logic. State machines therefore provide a uniform way to describe computation with simple mathematics.*

A state machine — and, note, we are not talking about *finite* state machines, but about state machines with possibly an infinite number of states that can describe any algorithm — is made of an initial set of states, a transition relation relating a current state to one or more next states, and possibly a fairness property that is important for demonstrating liveness, especially when specifying concurrent algorithms.

State machines are written in TLA+ in the *normal form* we've seen:

$$\exists\, w : Init \land \Box[Next]_v \land Fairness$$

Where $Init$ is the property, or condition, specifying the initial states, the $Next$ action specifies the next-state relation, $Fairness$ is the fairness property and the optional existential quantifier is used to hide internal state, as we'll see in part 4.

Very Simple State Machines

Nearly all of the effort of writing specifications — and most lines — goes into writing the next-state relation expressed by the $Next$ action of the normal form. Obviously, it's broken up into a hierarchy of pieces divided over many definitions — just like in programming — but the basic building blocks are simple.

Just as the values our variables can take are not the concern of TLA, but rather of the data logic, so too are the operations performed on those values, which form the primitive operations of our algorithms, namely, what transformations on the data can be performed at each state. In TLA+, any of the built-in or user-defined operators and functions we learned about in part 2 can serve as primitive operations.

Suppose x is a temporal variable. The action that says "if x is even, divide it by two", can be written as $x\%2 = 0 \land x' = x/2$. This is a common pattern. Note that it is *not* the same as the action $x\%2 = 0 \Rightarrow x' = x/2$

. The reason is that an action is a predicate on two states. In the first action, the formula is true — i.e. the primed state is a next-state of the unprimed state — iff x is even and $x' = x/2$. In the second action, if x is even and $x' = x/2$ then the predicate is indeed true, but it is also true if x is *not* even and x' is any value whatsoever, because an implication is true if its left operand is false; this is rarely what you want.

It is very common for an action to be a disjunction ("or") of conjuncts ("and"), like so:

$$\begin{aligned} C \triangleq \lor & \land x\%2 = 0 \\ & \land x' = x/2 \\ \lor & \land x\%2 \neq 0 \\ & \land x' = 3 * x + 1 \end{aligned}$$

In this particular case, the same action can be written equivalently as:

$$\begin{aligned} C \triangleq & \land x\%2 = 0 \Rightarrow x' = x/2 \\ & \land x\%2 \neq 0 \Rightarrow x' = 3 * x + 1 \end{aligned}$$

but this is only because $\forall\, x : x\%2 = 0 \lor x\%2 \neq 0$ so the left operand of one and only one of the $\Rightarrow$ connectives is guaranteed to be true at any time.

Of course, that action can also be equivalently written as:

$$\begin{aligned} C \triangleq \text{IF } x\%2 = 0 \text{ THEN } & x' = x/2 \\ \text{ELSE } & x' = 3 * x + 1 \end{aligned}$$

or as

$$C \triangleq x' = \text{IF } x\%2 = 0 \text{ THEN } x/2 \text{ ELSE } 3 * x + 1$$

We can now write the **Collatz conjecture** in TLA+:

$$\forall\, n \in Nat \setminus \{0\} : (x = n \land \Box[C]_x \land \mathrm{WF}_x(C)) \Rightarrow \Diamond(x = 1)$$

A property akin to (simple) type correctness in typed programming languages is modeled in TLA+ as a simple invariant (a state predicate that is asserted to hold in every state). Most TLA+ specifications contain a definition that looks like:

$$\begin{aligned} TypeOK \triangleq & \land x \in Int \\ & \land y \in Seq(Nat) \\ & \land z \in [name : \text{STRING}\ , ok : \text{BOOLEAN}\ ] \end{aligned}$$

Type preservation is then stated as a simple safety property: $Spec \Rightarrow \Box TypeOK$.

Nondeterminism

Nondeterminism plays a central role in TLA. Indeed, it is a crucial tool for analyzing programs in any formalism[25]. It can express unimportant detail (x is one of those values, doesn't matter which), IO (the user can enter any of these values), or concurrency (the scheduler may schedule this or that operation). But they all mean the same thing: we don't know or don't care about some details.

At the semantic level, nondeterminism is simply the existence of more than one behavior for a specification. In TLA+, nondeterminism is so crucial that *every* specification is nondeterministic with respect to the values of all variables that it *doesn't* mention; nondeterminism is the default, and that we must explicitly opt out of by writing something

we *do* know. Remember, all specifications exist in a single universe, and their model is all behaviors of *all possible* variables that satisfy a formula that mentions just a small finite set of them.

In the syntax, we can express nondeterminism in the initial condition, actions or any property, by writing propositions that are true for multiple values of a variable, using disjunction, set membership or existential quantification. For example, the following actions are all equivalent, and all forms are commonly found in real specifications:

$$\begin{aligned}
&[x' = 1 \lor x' = 2 \lor x' = 3]_x \\
\equiv\ &[x' \in 1..3]_x \\
\equiv\ &[\exists\, n \in 1..3 : x' = n]_x \\
\equiv\ &[\exists\, n \in Nat : n \geq 1 \land n \leq 3 \land x' = n]_x
\end{aligned}$$

Of course, "expressing nondeterminism" is a very bad choice of words, as that is really the exact opposite of what the expressions above do: they *restrict* nondeterminism. An empty formula is the same as TRUE: it allows all behaviors. In TLA, like in most logics, anything you write just reduces nondeterminism. $x' = 2$ reduces it a lot — it makes the choice of x completely deterministic, but it leaves all other infinity of variables deterministic; $x' \in 1..3$ reduces it just a bit less. The action $[x' = x + 1 \land y' = y]_{\langle x,y \rangle}$ (or $[x' = x + 1 \land \text{UNCHANGED } y]_{\langle x,y \rangle}$) says that at the next step, either x and y will not change, or that x will be incremented and y will not changed. The action $[x' = x + 1]_{\langle x,y \rangle}$, which is equivalent to $[x' = x + 1]_x$, says that at the next step, either x will not change or that x will be incremented; in either case, y (and z or any other variable) could be any value whatsoever. Not mentioning a variable at all is full nondeterminism. Once you mention it, it's just a matter of how much that nondeterminism is to be restricted.

Level of Detail

In part 1 I quoted Lamport **saying**:

> *… The obsession with language is a strong obstacle to any attempt at unifying different parts of computer science. When one thinks only in terms of language, linguistic differences obscure fundamental similarities.*

To see how programming languages obscure important similarities and highlight less important detail — necessary only for technical reasons — let's consider the following C/Java examples, adapted from *Computation and State Machines* (the paper from which the above quote was taken), for computing the factorial:

```
fact1(int n) { int f = 1;
               for (int i = 2; i <= n; i++)
                   f = f*i;
               return f; }

fact2(int n) { int f = 1;
               for (int i = n; i > 1; i--)
                   f = f*i;
               return f; }

fact3(int n) { return (n <= 1) ? 1 : n * fact3(n - 1); }
```

Most programmers will say that `fact1` and `fact2` are similar, because they're both iterative or imperative, whereas `fact3` is recursive or "functional". However, those are just similarities or differences in the style of *expressing* computations; they don't mean that the computations thus expressed are different. It is actually `fact2` that differs most from the other two, and it yields the same result only because multiplication is commutative. We could see that by replacing multiplication with a non-commutative operation like subtraction; the first and third program would give the same result, while the second would give a different one.

In TLA+, we would write the algorithms in state machines (I'll ignore fairness, which isn't necessary for safety properties). The algorithm carried out by `fact2` would be:

$$
\begin{array}{l}
\text{VARIABLES } N, i, f \\
\\
Fact2 \triangleq \land N \in Nat \land f = 1 \land i = N \\
\qquad\quad \land \Box[\text{IF } i > 1 \ \ \text{THEN } f' = f * i \land i' = i - 1 \land \text{UNCHANGED } N \\
\qquad\qquad\qquad\quad \text{ELSE } \ \text{UNCHANGED } \langle f, i, N\rangle]_{\langle f,i,N\rangle}
\end{array}
$$

Whereas the algorithm in `fact1` and `fact3` is:

$$
\begin{array}{l}
\text{VARIABLES } N, i, f \\
\\
Fact1 \triangleq \land N \in Nat \land f = 1 \land i = 2 \\
\qquad\quad \land \Box[\text{IF } i \leq N \ \text{THEN } f' = f * i \land i' = i + 1 \land \text{UNCHANGED } N \\
\qquad\qquad\qquad\quad \text{ELSE } \ \text{UNCHANGED } \langle f, i, N\rangle]_{\langle f,i,N\rangle}
\end{array}
$$

Of course, one may object that the use of recursion in `fact3` is not an irrelevant detail at all as it may have important consequences (say, on memory consumption), and so deserves a much more detailed specification, like this one:

$$
\begin{array}{l}
\text{VARIABLES } N, i, f, \\
\qquad\qquad pc, \quad \text{The program counter} \\
\qquad\qquad stack \quad \text{Contains tuples of variable values and the pc} \\
vars \triangleq \langle f, i, pc, stack\rangle \\
\\
Top \quad\ \ \triangleq stack[1] \\
Push(x) \triangleq stack' = \langle x\rangle \circ stack \\
Pop \quad\ \ \triangleq stack' = Tail(stack) \\
\\
Return(x) \ \triangleq f' = x \land \langle i, pc\rangle' = Top \land Pop \\
Recurse(x) \triangleq i' = x \land pc' = 1 \land Push(\langle i, pc + 1\rangle) \\
\\
Fact3 \triangleq \land N \in Nat \land f = 1 \land i = N \land pc = 1 \land stack = \langle\langle \text{“}- \text{”}, -1\rangle\rangle \\
\qquad\quad \land \Box[\land \ \text{IF } pc = -1 \text{ THEN UNCHANGED } vars \\
\qquad\qquad\quad \text{ELSE IF } pc = 1 \land i > 1 \text{ THEN } Recurse(i - 1) \land \text{UNCHANGED } f \\
\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } \ Return(\text{IF } i \leq 1 \text{ THEN } 1 \text{ ELSE } i * f) \\
\qquad\qquad \land \text{UNCHANGED } N]_{vars}
\end{array}
$$

After all, who decides which details are important and which are relevant? The answer is that you do, depending on what you want to model, and what you want to model depends on what properties of your system or algorithm you want to reason about. In any event, TLA+ gives the very notion of detail precise meaning, and, as we'll see in part 4,

$Fact3$ is an *implementation* of $Fact1$ — it is a more detailed (refined) description of $Fact1$ — but not of $Fact2$. Whether or not the extra detail is important, depends on what we want to do with it.

For example, modeling the stack may be important if we want to say something about the space complexity of the algorithm. First, we need to add a measure of space complexity:

$$\begin{array}{l}\text{VARIABLE } depth \\ Space \triangleq depth = 0 \land [depth' = depth + (Len(stack)' - Len(stack))]_{\langle depth, stack \rangle}\end{array}$$

We can now say something about the worst-case space complexity:

$$\forall\, n \in Nat : N = n \land Fact3 \land Space \Rightarrow \Box[depth \leq n]$$

Note how we restrict $Fact3$ to a single computation on a single input by conjoining it with $N = n$, which restricts $Fact3$'s initial condition $N \in Nat$.

Even though that's sufficient, let's do something a little more general (but probably less useful — my intent is to show the power of the formalism, not to teach best practices), that captures the exact definition of complexity:

$$BigO(f) \triangleq \{g \in [Nat \to Nat] : \exists\, c \in Nat : \forall\, k \in Nat : g[k] \leq c * f[k]\}$$

$$\begin{array}{l}Complexity(CompClass, Alg, Input, InputSize(\_), measure) \triangleq \\ \exists\, f \in CompClass : \forall\, x \in Input : Alg(x) \Rightarrow \Box(measure \leq f[InputSize(x)])\end{array}$$

Then, to express the fact that the space complexity of $Fact3$ is exponential (the size of the input is the logarithm of the integer argument), we can write:

$$\begin{array}{l}Complexity(BigO(k \in Nat \mapsto 2^k), \\ \qquad \text{LAMBDA } n : N = n \land Fact3 \land Space, \\ \qquad Nat, \text{LAMBDA } n : log(n), depth)\end{array}$$

However, as we'll see in part 4, $Fact1$ (and so $Fact3$ as well) and $Fact2$, are both implementations of the following specification of $FactAlg$,

$$\begin{array}{l}\text{VARIABLES } N, f \\ FactAlg \triangleq \text{LET } fact[n \in Nat] \triangleq \text{IF } n \leq 1 \text{ THEN } 1 \text{ ELSE } n * fact[n-1] \\ \qquad \text{IN } N \in Nat \land f = fact[N] \land \Box[\text{UNCHANGED } f]_f\end{array}$$

because they are extensionally equal (note that $fact$ is not an algorithm but a function).

State machines, then, give us a very general, mathematical framework of reasoning about algorithms. Unlike state machines you may be familiar with — Turing machines or finite-state automata — the state machines that are specified in TLA+ are completely abstract in the sense that a state can be any mathematical structure (definable in the data logic), and a primitive step, defined as an action, can be as limited or as powerful as we desire (and as the data logic allows).

Many other models of computation can also be easily described as state machines in TLA+. For example, **lambda calculus** (or any other **abstract rewriting system**) is normally thought of as a state machine — where each step is a beta-reduction — only when combined with an **evaluation strategy** such as call-by-name or call-by-value. However, even without a specific evaluation strategy, lambda calculus is easily described as a state machine in TLA+26 that is nondeterministic with respect to an evaluation strategy. In fact, we could do that to prove the **confluence** of lambda expressions.

Concurrency

In part 2 we saw how to model data at different levels of details, and in the previous section we saw how we can model a stack if we find it to be of importance. But how do we model processes or threads?

Suppose we have the following two specifications:

$$
\begin{aligned}
&\text{VARIABLE } x \\
&InitX \triangleq x = 0 \\
&NextX \triangleq x' = x + 1 \\
&SpecX \triangleq InitX \wedge \Box[NextX]_x \wedge \text{WF}_x(NextX)
\end{aligned}
$$

$$
\begin{aligned}
&\text{VARIABLE } y \\
&InitY \triangleq y = 0 \\
&NextY \triangleq y' = y + 1 \\
&SpecY \triangleq InitY \wedge \Box[NextY]_y \wedge \text{WF}_y(NextY)
\end{aligned}
$$

What does the specification $SpecX \wedge SpecY$ mean? Well, let's write it down, and do some basic manipulations on the formula:

$$
\begin{aligned}
SpecX \wedge SpecY &\equiv (InitX \wedge \Box[NextX]_x \wedge \text{WF}_x(NextX)) \wedge (InitY \wedge \Box[NextY]_y \wedge \text{WF}_y(NextY)) \\
&\equiv (InitX \wedge InitY) \wedge (\Box[NextX]_x \wedge \Box[NextY]_y) \wedge (\text{WF}_x(NextX) \wedge \text{WF}_y(NextY))
\end{aligned}
$$

So initially both x and y are 0. Then, every action must satisfy both $[NextX]_x$ and $[NextY]_y$, so at every step x is either incremented or stays the same *and* y is either incremented or stays the same. This means that every step increments either x, y, both, or neither (if neither, then it's a stuttering step with respect to $\langle x, y \rangle$). $SpecX \wedge SpecY$ specifies what is essentially two processes, $SpecX$ and $SpecY$ running concurrently.

By the way, note that because $SpecX$ and $SpecY$ are really the same process operating on different variables, we could have written a parameterized specification like so:

$$
\begin{aligned}
&Init(v) \triangleq v = 0 \\
&Next(v) \triangleq v' = v + 1 \\
&Process(v) \triangleq Init(v) \wedge \Box[Next(v)]_v \wedge \text{WF}_v(Next(v))
\end{aligned}
$$

$$
\begin{aligned}
&\text{VARIABLES } x, y \\
&Concurrent \triangleq Process(x) \wedge Process(y)
\end{aligned}
$$

In fact, we can take that a step further, and generalize this composition to any number of processes:

$$
\begin{aligned}
&\text{CONSTANT } Counters \\
&\text{ASSUME } Counters \in Nat \\
&\text{VARIABLE } x \quad \in [1..Counters \to Nat], \text{ A sequence of counters} \\
&Many \triangleq \forall\, c \in Counters : Process(x[c])
\end{aligned}
$$

But for simplicity, we'll continue with the non-parameterized version, and take a closer look at the action part of the specification, now repeating the reasoning we did before using syntactic manipulation:

$$
\begin{aligned}
&\Box[NextX]_x \land \Box[NextY]_y \\
&\equiv \Box([NextX]_x \land [NextY]_y) \text{ Because } \Box(F \land G) \equiv (\Box F) \land (\Box G) \\
&\text{The followng are some informal steps containing expressions that aren't well-formed in TLA} \\
&\equiv \Box(\ \land NextX \lor x' = x \\
&\qquad \land NextY \lor y' = y) \qquad \text{Because of the definition of } [\ldots]_x \text{ and } [\ldots]_y \\
&\equiv \Box(\ \lor NextX \land NextY \qquad \text{Because of propositional logic equivalence} \\
&\qquad \lor NextX \land y' = y \\
&\qquad \lor NextY \land x' = x \\
&\qquad \lor x' = x \land y' = y) \\
&\text{Back to TLA} \\
&\equiv \Box[\ \lor NextX \land NextY \\
&\qquad \lor NextX \land y' = y \\
&\qquad \lor NextY \land x' = x]_{\langle x,y \rangle}
\end{aligned}
$$

We can therefore define:

$$
\begin{aligned}
Init &\triangleq \land x = 0 \\
&\quad\ \land y = 0 \\
Next &\triangleq \lor NextX \land NextY \\
&\quad\ \lor NextX \land y' = y \\
&\quad\ \lor NextY \land x' = x
\end{aligned}
$$

$$
Spec \triangleq Init \land \Box[Next]_{\langle x,y \rangle} \land \mathrm{WF}_x(NextX) \land \mathrm{WF}_y(NextY)
$$

Where $Spec$ is equivalent to $SpecX \land SpecY$.

Can we also combine $\mathrm{WF}_x(NextX) \land \mathrm{WF}_y(NextY)$ into $\mathrm{WF}_{\langle x,y \rangle}(Next)$? Absolutely not! The latter specifies that if $Next$ is continually enabled, it will eventually occur — and as $Next$ is always enabled, it means that it will occur infinitely often — which will only guarantee that *either* x or y will be incremented infinitely often, but maybe only x. If our processes are scheduled by a fair scheduler, then *both* x *and* y will be incremented infinitely often, which is what $\mathrm{WF}_x(NextX) \land \mathrm{WF}_y(NextY)$ specifies.

If we'd like to forbid a simultaneous change to both x and y to get an *interleaving specification* we could remove the first disjunct ($\lor NextX \land NextY$) from $Next$. This would result in $Spec$ being equivalent to:

$$
SpecX \land SpecY \land \Box[x' = x \lor y' = y]_{\langle x,y \rangle}
$$

The last conjunct forbids the simultaneous change to bot x and y.

The *conjunction* of the two specifications denotes a parallel composition of processes, but it is equivalent to what appears like a specification of a single process with a nondeterministic action, specified as a *disjunction*.

Lamport **writes**:

> *The idea that the execution of a sequential algorithm can be described as a sequence of states seems to be due to Turing. It is so widely accepted that most engineers are happy to describe sequential algorithms this way... I think the idea that the execution of a concurrent algorithm (or a concurrent*

> *system) can also be described by a sequence of states was implicit in the first paper on concurrent algorithms: Dijkstra's seminal 1965 paper on mutual exclusion. That idea is not widely known, and many engineers need to be shown how to describe concurrent systems in that way. When they learn how to write such descriptions in TLA⁺, they find them quite useful.*

In other words, there is no difference between multiple processes and a single process with a certain kind of nondeterminism. This has important consequences when reasoning about algorithms, because it means we can use the same proof techniques for both sequential and concurrent algorithms. Lamport's paper, ***Processes are in the Eye of the Beholder*** explores this idea further.

When writing specifications of concurrent systems, instead of writing conjunctions of specifications of each process, we normally write a single $Next$ action, which is a *dis*junction of sub-actions representing different processes. As we've seen, the two forms are logically equivalent, but if we care about liveness we must remember to specify weak (or strong) fairness of each sub-action. Here's an example:

$$
\begin{aligned}
&Init \triangleq \ldots \\
&A \triangleq \ldots \\
&B \triangleq \ldots \\
&Next \triangleq A \lor B \\
&Spec \triangleq Init \land \Box[Next]_{vars} \land \mathrm{WF}_{vars}(A) \land \mathrm{WF}_{vars}(B)
\end{aligned}
$$

Notice how $A \Rightarrow Next$ and $B \Rightarrow Next$, and so our fairness condition $\mathrm{WF}_{vars}(A) \land \mathrm{WF}_{vars}(B)$ satisfies the condition given in the section **Machine Closure and Fairness** that guarantees that our specification is machine-closed.

It is often the case that we have multiple instances of processes (or machines) each running the same program. We can model that like so:

$$
\begin{aligned}
&\text{VARIABLES } state \\
&Processes \triangleq 1..10 \\
&Init \triangleq state = [p \in Processes \mapsto \ldots] \\
&Run(self) \triangleq \ldots \\
&Next \triangleq \exists\, p \in Processes : Run(p) \\
&Fairness \triangleq \forall\, p \in Processes : \mathrm{WF}_{state}(Run(p)) \\
&Spec \triangleq Init \land \Box[Next]_{state} \land Fairness
\end{aligned}
$$

Note how the existential quantifier is used to nondeterministically select one of the processes. This works because actions are transition predicates. Let's see that with a concrete example:

$$
\begin{aligned}
&\text{VARIABLE } x \\
&Processes \triangleq 1..4 \\
&Init \triangleq x = [p \in Processes \mapsto 0] \\
&Run(self) \triangleq x' = [x \text{ EXCEPT } ![self] = @ + 1] \\
&Next \triangleq \exists\, p \in Processes : Run(p) \\
&Fairness \triangleq \forall\, p \in Processes : \mathrm{WF}_{x}(Run(p)) \\
&Spec \triangleq Init \land \Box[Next]_{x} \land Fairness
\end{aligned}
$$

Now, the transition $\langle 7, 5, 8, 19 \rangle \rightharpoonup \langle 7, 6, 8, 19 \rangle$ is a possible transition, because *there exists* a $p \in Processes$ — in this case $p = 2$ — such that $x' = [x \text{ EXCEPT } ![self] = @ + 1]$. Does this specification require interleaving or

does it allow multiple processes to run at once? After all, an expression with an existential quantifier may be true if more than one element satisfies the predicate. This example does not allow multiple processes to increment their counter simultaneously because the expression $x' = [x \text{ EXCEPT } ![self] = @ + 1]$ states that just one index in the sequence is changed, and cannot be true for two different values of $self$. Again, notice that for every $p \in Processes, Run(p) \Rightarrow Next$ and so our fairness condition satisfies the condition in the **Machine Closure and Fairness**, and so our specification is machine-closed.

In general, for an interleaving specification (which is what we want most of the time) under normal conditions, the composition of two specifications $A \land B$ becomes $(InitA \land InitB) \land \Box[NextA \lor NextB]_v \land (\text{WF}_{vars}(A) \land \text{WF}_{vars}(B))$ – i.e., the conjunction of the temporal formula becomes a disjunction in the action – and similarly, a composition of an arbitrary number of specifications, $\forall i \in Component : P(i)$, becomes $(\forall i \in Component : InitP(i)) \land \Box[\exists i \in Component : NextP(i)]_v$ – i.e., the universal quantification in temporal formula becomes an existential quantification in the action.

Concurrency is yet another example where programming languages emphasize differences that are important details when the goal is efficient compilation but unimportant for reasoning, and obscure similarities that help with reasoning about algorithms. Because a composition of specifications is equivalent to a specification with a single action, it is in general easier to reason about (but not to *program*!) a complex system when it is expressed with a single action. Lemport makes this argument in his paper, ***Composition: A Way to Make Proofs Harder***.

Let's now consider a more complicated example of concurrency. We have a producer adding elements to a FIFO queue, and a consumer removing them:

$$
\begin{aligned}
&\text{VARIABLE } buf \\
&\\
&\text{VARIABLE } put \\
&pvars \triangleq \langle buf, put \rangle \\
&ProducerInit \triangleq buf = \langle\rangle \\
&ProducerNext \triangleq \land Len(buf) < 10 \\
&\qquad\qquad\quad \land put' \in Int \\
&\qquad\qquad\quad \land buf' = Append(buf, put) \\
&ProducerSpec \triangleq ProducerInit \land \Box[ProducerNext]_{pvars} \land \text{WF}_{pvars}(ProducerNext) \\
&\\
&\text{VARIABLE } get \\
&cvars \triangleq \langle buf, get \rangle \\
&ConsumerInit \triangleq buf = \langle\rangle \\
&ConsumerNext \triangleq \land buf \neq \langle\rangle \\
&\qquad\qquad\quad \land get' = Head(buf) \\
&\qquad\qquad\quad \land buf' = Tail(buf) \\
&ConsumerSpec \triangleq ConsumerInit \land \Box[ConsumerNext]_{cvars} \land \text{WF}_{cvars}(ConsumerNext)
\end{aligned}
$$

What happens when we conjoin the two specifications? Clearly, the initial condition poses no problem because $ProducerInit \land ConsumerInit \equiv buf = \langle\rangle$. But $ProducerNext$ says that at every step buf must be appended to or not change, and $ConsumerNext$ says that at every step buf must have an element removed or not change. Formally:

$$
\begin{aligned}
&\Box[ProducerNext]_{p}vars \land \Box[ConsumerNext]_{p}vars \\
\equiv\ &\Box([ProducerNext]_{x} \land [ConsumerNext]_{y}) \\
&\text{Not TLA:} \\
\equiv\ &\Box(\ \land ProducerNext \quad \lor (buf' = buf \land put' = put) \\
&\qquad \land ConsumerNext \lor (buf' = buf \land get' = get)) \\
\Rightarrow\ &\Box(\ \lor ProducerNext \land ConsumerNext \\
&\qquad \lor buf' = buf \land get' = get \land put' = put) \\
&\text{Back to TLA} \\
\equiv\ &\Box[ProducerNext \land ConsumerNext]_{\langle buf, put, get \rangle} \\
\Rightarrow\ &\Box[\ \land buf' = Append(buf, put) \\
&\qquad \land buf' = Tail(buf)]_{\langle buf, put, get \rangle} \\
\Rightarrow\ &\Box[\ Append(buf, put) = Tail(buf)]_{\langle buf, put, get \rangle} \\
\Rightarrow\ &\Box[\text{FALSE}]_{\langle buf, put, get \rangle}
\end{aligned}
$$

The result is that $ProducerSpec \land ConsumerSpec$ allows only behaviors where buf never changes. The two specifications must be made more flexible to allow each other to change their shared state. We'll make the following adaptations:

$$
\begin{aligned}
ProducerNext1 \triangleq\ &\lor ProducerNext \\
&\lor\ \land \text{UNCHANGED } put \\
&\quad\ \land buf' = Tail(buf)
\end{aligned}
$$

$$
\begin{aligned}
ConsumerNext1 \triangleq\ &\lor ConsumerNext \\
&\lor\ \land \text{UNCHANGED } get \\
&\quad\ \land\ \exists x : buf' = Append(buf, x)
\end{aligned}
$$

Notice how each action protects its own "private" variable (put/get), yet allows modifications to the shared variable buf, provided that they comply with its behavior, namely adding or removing an element at a time. Now

$$[ProducerNext1]_{pvars} \land [ConsumerNext1]_{cvars} \equiv [ProducerNext \lor ConsumerNext]_{\langle get, put, buf \rangle}$$

From the perspective of the consumer (and similarly, from that of the producer), when we've added the disjunct in $ConsumerNext1$ we've really just used nondeterminism to model *input*. We didn't place any restrictions on the input, while the actual producer only puts integers in the queue; a producer that produces strings would have worked as well. You can think of IO, concurrency and nondeterminism as different ways of looking at the same thing. An interactive algorithm is a concurrent algorithm; whether the user's behavior is modeled as a separate process or as "inline" nondeterminism within a single action is just a matter of notation.

Lamport **writes**:

> *Mathematical manipulation of specifications can yield new insight. A producer/consumer system can be written as the conjunction of two formulas, representing the producer and consumer processes. Simple mathematics allows us to rewrite the same specification as the conjunction of n formulas, each representing a single buffer element. We can view the system not only as the composition of a producer and a consumer process, but also as the composition of n buffer-element processes. Processes are not fundamental components of a system, but abstractions that we impose on it. This insight could not have come from writing specifications in a language whose basic component is the process.*

Plotkin's Parallel Or

Using logical conjunction to denote the parallel composition of concurrent components — or of a system component with input from the environment — is just a special case of composition. We will explore composition more generally in part 4, but as an exercise, let's consider the following problem: how do we specify a program that takes two programs as arguments, and terminates iff one of them terminates. This problem is known as *Plotkin's parallel or*, after Gordon Plotkin, who studied it in the context of **reduction strategies** for programming languages. The general idea is to simulate each program for a single step in an interleaved fashion.

We can do this without requiring each program's "source code" to be represented as data, even though programs in TLA are expressed as opaque formulas, and we have no mechanism (like Lisp's macros), to **quote** them and access their internal structure:

$$
\begin{array}{l}
ParallelOr(Spec0, state0, Spec1, state1) \triangleq \\
\quad \text{LET } Alternator \triangleq \\
\qquad \exists\, s : \land\, s = 0 \\
\qquad\qquad \land\, \Box[\, \lor\, s = 0 \land s' = 1 \land state0' \neq state0 \land state1' = state1 \\
\qquad\qquad\qquad \lor\, s = 1 \land s' = 0 \land state0' = state0 \land state1' \neq state1]_{\langle s, state0, state1 \rangle} \\
\quad \text{IN } Spec0 \land Spec1 \land Alternator
\end{array}
$$

where $Spec0$ and $Spec1$ are the formulas specifying the two programs (for simplicity, we're assuming they don't have liveness conditions[27]), and $state0$ and $state1$ are the tuples of temporal variables used in each, respectively (assuming that they don't share variables; in part4 we'll see how we can rename variables so that we can enforce this). The internal variable s, introduced with the temporal existential quantifier, serves to alternate between one program to the other. As soon as one program terminates — i.e., no longer changes its state — the switching mechanism cannot switch to the other program, and so gets stuck, and the combined program terminates as well.

The ability to do this without quoting or encoding code as data in any way, stems directly from our choice of representing programs as dynamical systems rather than as a mapping of inputs to outputs.

Also, note how $Alternator$, which would be a higher-order combinator in other formalisms, as it "operates" on other programs, is composed with them in a very first-order way (plus,the composition is, obviously, completely commutative: we could also equivalently write $Spec0 \land Alternator \land Spec1$ or $Alternator \land Spec0 \land Spec1$). In part 4 we'll see why this form of composition through conjunction is so general.

More Complex Examples

While I won't describe how to write a large, complex specification (in part 1 I listed some resources containing tutorials and examples) — and so you'll have to take my word, and Amazon's technical report I mentioned in part 1, that TLA$^+$ specifications scale very gracefully to large, complex systems — I do want to show a couple of more complex example than the ones we've seen, algorithms that actually do something interesting. Being able to easily read and understand these specifications does take a bit of practice, but my intention here is just to give a sense of how a real program can be described as a logical formula

I've chosen two examples given by Lamport, both of sorting algorithms. The specifications will make use of some of the definitions we've defined in part 2, such as $Toset$, $Permutations$, $Ordered$ and $Image$. We'll assume that both specifications are preceded by the following declarations and definitions, which establish that S is some set with a partial order defined on it:

$$\begin{array}{l}\text{CONSTANTS } S, \_ \preceq \_ \\ \text{ASSUME } Toset(S, \preceq)\end{array}$$

The first example is an exchange sort (taken from **this talk** by Lamport). Informally, the algorithm is this: as long as the array is not sorted, pick a pair of consecutive elements that are out of order and swap them. Here it is in TLA+:

$$\begin{array}{l}\text{VARIABLES } A, A0, done \\ vars \triangleq \langle A, A0, done \rangle \\ \\ Init \triangleq A \in Seq(S) \land A0 = A \land done = \text{FALSE} \\ Next \triangleq \text{IF } \neg Ordered(A, \preceq) \text{ THEN} \\ \qquad \land \exists p \in \{q \in 1..(Len(A) - 1) : \neg(A[q] \preceq A[q+1])\} : \\ \qquad\qquad A' = [A \text{ EXCEPT } ![p] \quad = A[p+1], \\ \qquad\qquad\qquad\qquad ![p+1] = A[p]] \\ \qquad \land \text{UNCHANGED } \langle A0, done \rangle \\ \qquad \text{ELSE } done' = \text{TRUE } \land \text{UNCHANGED } \langle A, A0 \rangle \\ ExchangeSort \triangleq Init \land \Box[Next]_{vars} \land \text{WF}_{vars}(Next) \\ \\ \text{THEOREM } ExchangeSort \Rightarrow \Box(done \Rightarrow A \in Permutations(A0) \land Ordered(A, \preceq)) \\ \text{THEOREM } ExchangeSort \Rightarrow \Diamond done\end{array}$$

Note how high-level our description is. We don't specify how we find the pair to swap or how we tell that the array is not yet sorted. Of course, we could add those details, but this level of description captures very clearly the gist of how exchange-sort works.

The next example is Quicksort. For a step-by-step explanation of how the specification is written, see **this explanation by Lamport**. This specification is also at a very high level, attempting to describe only the essence of Quicksort and little else. It describes Quicksort like so: maintain a set of unsorted array regions; remove a region from the set and, if it's not empty, partition it around some pivot point so that it now contains two sub-regions, the left with elements smaller or equal to the elements in the right; add those two sub-regions to the set.

For the partitioning operation, the spec relies on the following mathematical definition of the set of all possible legal partitions, i.e. the set of sequences obtained from B by permuting $B[low]..B[hi]$ such that the subarray left of $lo \leq pivot < hi$ contains only elements that are smaller than in the subarray on the right:

$$\begin{array}{l}Parititions(B, pivot, lo, hi) \triangleq \\ \qquad \{SubSeq(B, 1, lo-1) \circ s \circ SubSeq(B, hi+1, Len(B)) : \\ \qquad\qquad s \in \{s \in Permutations(SubSeq(B, lo, hi)) : \\ \qquad\qquad\qquad \text{LET } piv \triangleq pivot - low + 1 \text{ IN} \\ \qquad\qquad\qquad \forall\, x \in Image(SubSeq(s, 1, piv)), y \in Image(SubSeq(s, piv, Len(s))) : x \preceq y\}\}\end{array}$$

Here is the spec:

$$
\begin{array}{l}
\text{VARIABLES } A, A0, U, done \\
vars \triangleq \langle A, A0, U, done \rangle \\
\\
Init \triangleq \land A \in Seq(S) \\
\qquad\quad \land U = \{\langle 1, Len(A) \rangle\} \quad \text{A set of regions on which Partition needs to be called} \\
\qquad\quad \land A0 = A \land done = \text{FALSE} \\
Next \triangleq \text{IF } U \neq \{\} \text{ THEN} \\
\qquad\qquad \land \exists \langle a, b \rangle \in U : \\
\qquad\qquad\qquad \text{IF } a < b \\
\qquad\qquad\qquad\quad \text{THEN } \land \exists pivot \in a..b-1 : A' \in Partitions(A, pivot, a, b) \\
\qquad\qquad\qquad\qquad\quad\ \land U' = (U \setminus \{\langle a, b \rangle\}) \cup \{\langle a, pivot \rangle, \langle pivot + 1, b \rangle\} \\
\qquad\qquad\qquad\quad \text{ELSE } \land \text{UNCHANGED } A \\
\qquad\qquad\qquad\qquad\quad\ \land U' = U \setminus \{\langle a, b \rangle\} \\
\qquad\qquad \land \text{UNCHANGED } \langle A0, done \rangle \\
\qquad\quad \text{ELSE } done' = \text{TRUE } \land \text{UNCHANGED } \langle A, A0, U \rangle \\
QuickSort \triangleq Init \land \Box[Next]_{vars} \land \text{WF}_{vars}(Next) \\
\\
\text{THEOREM } QuickSort \Rightarrow \Box(done \Rightarrow A \in Permutations(A0) \land Ordered(A)) \\
\text{THEOREM } QuickSort \Rightarrow \Diamond done
\end{array}
$$

Note how the specification makes no mention of recursion in the familiar programming sense — that of a subroutine calling itself, storing data on the stack, and yet captures the essence, the partitioning around a pivot, without specifying how exactly the problem is divided in order to conquer.

This specification makes use of two different forms of "ignoring detail". One is the use of nondeterminism in the choice of the region to sort and the particular partition; the other is the specification of the partitioning operation as a mathematical definition using the data logic rather than as a computation.

It is easy to see how nondeterminism is lack of detail in a precise way. For example, because $x' = 1 \Rightarrow x' \in \{0, 1\}$, clearly $(x = 0 \land \Box[x' = 1]_x) \Rightarrow (x = 0 \land \Box[x' \in \{0, 1\}]_x)$. But in what precise sense is the use of a mathematical definition rather than a computation "ignoring detail"? We will answer that question in part 4.

However, you may rightly object that the specification does not exactly match my description of Quicksort in the section **Algorithms and Programs** as it performs the partitions sequentially, while doing it in parallel is essentially allowed by the algorithm. To specify the algorithm precisely, instead of picking just one pair from U at each step we nondeterministically pick any number of pairs at once. Doing so requires just a little bit more work, but I've chosen to keep the presentation sequential so that it would be a little easier to read for those new to TLA+.

Some algorithms — especially those that are specified at the code level — may be easier for programmers to specify in PlusCal, a pseudocode-like language that can be written inside a comment block in a TLA+ file, and automatically translated to a readable TLA+ specification. While I will show a tiny example of PlusCal later, I will not cover it, as it is little more than syntax sugar around TLA+. The resources I linked to in part 1 teach PlusCal (except for *Specifying Systems*, which had been written prior to the development of PlusCal).

What We Can and Cannot Specify

Once we have a system specification, we make statements about its behavior that are normally of the form:

$$\text{THEOREM } Spec \Rightarrow P$$

i.e., $\vdash Spec \Rightarrow P$, Where P can be a property that our specification is supposed to satisfy, or another system specification it implements (we will talk at length about relations on algorithms in TLA in part 4).

What kind of properties do we want to verify? Very often those will be simple safety properties, of the kind $\Box I$, where I is a state predicate, for example, partial correctness, like $\Box(done \Rightarrow Ordered(A))$ or $\exists\, c \in Nat : \Box(x + y = c)$. Such properties are called *invariants*. We can also make some more interesting statements. For example, $\Box[x' > x]_x$, which states that the value of the variable x can only increase.

We can also write liveness properties like termination, $\Diamond done$, or, "for every request there will eventually be a response" – $\forall\, n \in Nat : \Box(request.id = n \Rightarrow \Diamond(response.id = n))$. This can be written as $\forall\, n \in Nat : request.id = n \rightsquigarrow response.id = n$. However, we'll probably model $request$ and $response$ as sequences to which requests and responses are being added so:

$$\forall\, n \in Nat : (\exists\, req \in Image(request) : req.id = n) \rightsquigarrow (\exists\, res : Image(response) : res.id = n)$$

We can also express more complex properties like worst-case complexity, as we've seen above. In practice, we'd just add a counter variable directly to the specification, but as we're dealing with theory, we can write:

$$\begin{array}{l} \text{VARIABLE } time \\ Time \triangleq time = 0 \land [time' = time + 1]_{\langle time, vars \rangle} \end{array}$$

which increments $time$ whenever any of the specification's variables change, and then something like:

$$\begin{array}{rl} \exists\, c \in Nat : Spec \land Time \Rightarrow & \text{LET } n \triangleq Size(input) \text{ IN} \\ & \Box(time \leq n^2) \end{array}$$

to state that the worst-case time complexity is quadratic (compare with the even more general complexity operator we defined **above**).

However, because TLA is a first-order logic, any property we write in a formula must be true for *all* behaviors of the specification. Remember $\vdash F$, iff *all* behaviors satisfy F. This works for all safety and liveness properties, but we can't, for example, express a property about the *average* complexity of an algorithm, as that would be a property on the *set* of behaviors of the algorithm, and not of each and every one; in other-words, average-case complexity is a second-order property.

Similarly, we cannot express as a TLA formula a (wrong) statement like: for every property, if two different algorithms satisfy the property, then they must share at least one behavior. It is questionable whether such statements are of any interest at all to engineers (probably not), and in any case, we can express second-order propositions not as formulas but as an ASSUME/PROVE construct (which we've seen in part 2, and will see in the context of TLA in the next section).

It is possible to specify a probabilistic machine in TLA+ like so:

$$I(a, b) \triangleq \{x \in Real : x \geq a \land x < b\} \quad \text{Half-open interval}$$

$$\begin{array}{rl} Next \triangleq \exists\, p \in I(0,1) : \land\, p \in I(0, 0.5) & \Rightarrow A \\ \land\, p \in I(0.5, 0.75) & \Rightarrow B \\ \land\, p \in I(0.75, 1) & \Rightarrow C \end{array}$$

where A, B and C are actions. However, it is unfortunately impossible to specify a probabilistic property, such as "x is greater than 0 with probability 0.75". This is, I think, the one glaring omission of the formalism. It seems like it may be possible to add this ability to TLA while keeping it first-order with the addition of a single probability

operator, as described in a paper by Zoran Ognjanović, **Discrete Linear-time Probabilistic Logics: Completeness, Decidability and Complexity** (2006).

Invariants and Proofs

TLA has a *relatively complete* proof theory for formulas written in the normal form. If F and G are such formulas and $F \Rightarrow G$ is valid, i.e., $\models F \Rightarrow G$, then it is provable by the TLA proof rules, i.e., $\vdash F \Rightarrow G$, provided that the necessary assertions about the actions in F and G — written in the data logic — can be proven[28].

Because TLA is a temporal logic, it suffers from some of the shortcomings of temporal logic. In ordinary logic, $(F \vdash G) \vdash (F \Rightarrow G)$. However, this is not true in temporal logic (or in any modal logic for that matter). If F is a temporal formula, then $\vdash F$ iff F is true for all behaviors (via a completeness theorem for temporal logic). $F \vdash G$ means that if F is true for all behaviors, then it is provable that so is G. This means that the proof rule $F \vdash \Box F$ is valid because if F is true of all behaviors, then it is also true for all suffixes of all behaviors (because a suffix of a behavior is a behavior). But $F \not\Rightarrow \Box F$, or otherwise, $x = 1 \Rightarrow \Box(x = 1)$, which is not true. So in temporal logic, $(F \vdash G) \not\vdash (F \Rightarrow G)$. This happens because implication, like all the logical connectives, "lives" in a modality, whereas $\vdash$ talks about *all* modalities.

Because of this, Lamport **writes**:

> *[This] principle lies at the heart of all ordinary mathematical reasoning. That it does not hold means temporal logic is evil. Unfortunately, it is a necessary evil because it's the best way to reason about liveness properties. TLA is the least evil temporal logic I know because it relies as much as possible on non-temporal reasoning.*

As with the data logic, we can write second-order theorems to serve as proof rules for TLA using the ASSUME /PROVE construct. In part 2 we learned that the CONSTANT keyword (which is synonymous with NEW CONSTANT or just NEW) introduces an arbitrary constant (level-0) name. Similarly, we can introduce names referring to state, action and temporal expressions with STATE (or NEW STATE), ACTION (or NEW ACTION), or TEMPORAL (or NEW TEMPORAL) respectively. Remember that CONSTANT, STATE, ACTION and TEMPORAL form a hierarchy, where an object of one kind is also an instance of all following kinds. We can then state second-order theorems like the following:

$$\begin{aligned} \text{THEOREM ASSUME TEMPORAL } F,\ &\text{TEMPORAL } G,\ \text{TEMPORAL } H, \\ & F \leadsto G,\ G \leadsto H \\ \text{PROVE} \quad & F \leadsto H \end{aligned}$$

While TLA has various proof rules, some of which apply to liveness like the one above, we will not concern ourselves with proving liveness at all but only with safety properties. In fact, we will not concern ourselves with any of the temporal proof rules but one.

As a verification tool, TLA belongs in the family of methods making use of *assertional reasoning*, which means that we assert that certain propositions are true at various states of the program. Another forms of reasoning include *behavioral reasoning*, which makes statements about the program's entire behavior[29].

It is easy to reason behaviorally in an assertional framework by adding a variable which holds a sequence of all the program's states so far; at each step, the new state is added to the sequence. Such a variable is called a history variable.

Lamport **writes**:

> *What a program does next depends on its current state, not on its history. Therefore, a proof that is based on a history variable doesn't capture the real reason why a program works. I've always found that proofs that don't use history variables teach you more about the algorithm.*

The best known assertional reasoning method for sequential algorithms is the **Floyd-Hoare method**, that attaches an assertion to each of the programs control points (lines). Those assertions are commonly written as *Hoare triplets*, PCQ, where C is a program statement, P is an assertion of the *precondition*, namely a proposition about the state of the program before the execution of C, and Q is a *postcondition*, a proposition about the state of the program after the execution of P. The Floyd-Hoare method is used to prove what's known as *partial correctness*, namely a property of the program that holds true if and when the program terminates.

If $\vdash Spec \Rightarrow \Box P$ then P is said to be an *invariant* of the program — something that holds true at every step. As we've seen, partial correctness is an invariant of the form $Terminated \Rightarrow P$.

In general, a Hoare triplet can be written in TLA as $P \land C \Rightarrow Q'$, where P and Q are state predicates expressing pre- and post-conditions respectively, and C is an action. In TLA, a program's control points are made explicit, using a variable that we can call pc (for program counter) that corresponds with a label or a line of the program. If our program contains a statement $c : C; d : \ldots$ where c and d are labels or line numbers, we can write the Hoare triplet as $(pc = c) \land (pc' = d) \land P \land C \Rightarrow Q'$. Looking at the program as a whole, the collection of all Hoare triplets can be expressed as the invariant:

$$\forall\, c \in ControlPoint : (pc = c) \Rightarrow P(c)$$

Where $P(c)$ is presumably written as:

$$\begin{aligned} P(c) \triangleq \land\, & c = c1 && \Rightarrow Assertion1 \\ \land\, & c = c2 && \Rightarrow Assertion2 \\ & \ldots \\ \land\, & c = \text{“done ”} && \Rightarrow PartialCorrectness \end{aligned}$$

where the assertions do not mention the variable pc (as Hoare triplets cannot).

The Floyd-Hoare method is *relatively complete* for simple programming languages, i.e., if the individual assertions can be proven, then so can partial correctness, but it is not relatively complete for languages with procedures, because partial correctness may require proof that makes use of the state of the call-stack, which the Floyd-Hoare method cannot directly mention.

The Floyd-Hoare method has been generalized for concurrent programs by the Owicki-Gries method, which attaches predicates for each control point *of each process*. Similarly to the Floyd-Hoare method, Owicki-Gries assertions can be written in TLA as:

$$(pc[p] = c) \Rightarrow P(p, c)$$

where c is a control point and p is a process (recall how we modeled multiple processes in the section **Concurrency** above).

In a note about his 1977 paper, *Proving the Correctness of Multiprocess Programs*, Lamport **writes**:

> *In the mid-70s, several people were thinking about the problem of verifying concurrent programs. The seminal paper was Ed Ashcroft's Proving Assertions About Parallel Programs, published in the Journal of Computer and System Sciences in 1975. That paper introduced the fundamental idea of invariance. I discovered how to use the idea of invariance to generalize Floyd's method to multiprocess programs. As is so often the case, in retrospect the idea seems completely obvious. However, it took me a while to come to it. I remember that, at one point, I thought that a proof would require induction on the number of processes.*
>
> *When we developed our methods, Owicki and I and most everyone else thought that the Owicki-Gries method was a great improvement over Ashcroft's method because it used the program text to decompose the proof. I've since come to realize that this was a mistake. It's better to write a global invariant. Writing the invariant as an annotation allows you to hide some of the explicit dependence of the invariant on the control state. However, even if you're writing your algorithm as a program, more often than not, writing the invariant as an annotation rather than a single global invariant makes things more complicated. But even worse, an annotation gets you thinking in terms of separate assertions rather than in terms of a single global invariant. And it's the global invariant that's important. Ashcroft got it right. Owicki and Gries and I just messed things up. It took me quite a while to figure this out.*

Because TLA allows us to reason globally about the program's state, there is no need to use just invariants of the forms above — those are appropriate for reasoning about programs written in a programming language — rather, we can generalize assertional reasoning completely. That reasoning would work equally well for concurrent and sequential programs, as they are the same kind of object in our model (processes are in the eye of the beholder etc.). A general proof method that works well for many kinds of algorithms, is the inductive invariant method, which works as follows.

Suppose we have a specification $Spec \triangleq Init \land \Box[Next]_v \land Fairness$, and want to prove some invariant (safety property) P — i.e. $Spec \Rightarrow \Box P$ — like partial correctness. We find a state predicate, Inv, such that $Inv \Rightarrow P$, but also $Init \Rightarrow Inv$ and $Inv \land Next \Rightarrow Inv'$. In other words, Inv holds in the initial states, and it is preserved by state transitions. Inv is then called an *inductive invariant*. We make use of the following proof rule:

$$\begin{aligned} &\text{THEOREM ASSUME ACTION } Next, \text{ STATE } v, \text{ STATE } Inv, \\ &\qquad\qquad Inv \land [Next]_v \Rightarrow Inv' \\ &\qquad \text{PROVE} \quad Inv \land \Box[Next]_v \Rightarrow \Box Inv \end{aligned}$$

If $Init \Rightarrow Inv$, using the proof rule above we can conclude that $Init \land \Box[Next]_v \Rightarrow \Box Inv$. In other words, if Inv is an inductive property, then it is an invariant. And since it's an invariant and $Inv \Rightarrow P$ then so is P, and we conclude $Spec \Rightarrow \Box P$.

Formal TLA$^+$ proofs based on inductive invariants all have the following structure:

$$
\begin{array}{l}
\text{THEOREM } Spec \Rightarrow \Box P \\
\quad \langle 1 \rangle 1.\ Inv \Rightarrow P \\
\qquad \ldots \\
\quad \langle 1 \rangle 2.\ Init \Rightarrow Inv \\
\qquad \ldots \\
\quad \langle 1 \rangle 3.\ Inv \wedge [Next]_{vars} \Rightarrow Inv' \\
\qquad \ldots \\
\quad \langle 1 \rangle \text{ QED} \\
\qquad \text{BY } \langle 1 \rangle 1, \langle 1 \rangle 2, \langle 1 \rangle 3, PTL \text{ DEF } Spec
\end{array}
$$

Where PTL stands for propositional temporal logic, and means use of temporal deduction rules. Specifically, the mechanical proof system TLAPS uses a solver for propositional temporal logic to checks proofs by PTL.

Let's look at a very simple example. Suppose our algorithm is:

$$
\begin{array}{l}
\text{VARIABLE } x \\
Init \triangleq x \in \{0, 1\} \\
Next \triangleq x' = 1 - x \\
Spec \triangleq Init \wedge \Box[Next]_x \wedge \text{WF}_x(Next)
\end{array}
$$

The safety property we wish to prove would also be a simple one:

$$P \triangleq x \geq 0$$

While P is an invariant (which we can conclude by observation in this trivial case), unfortunately it is not an inductive invariant because $P \wedge Next \nRightarrow P'$, for example if $x = 3$. Instead, the inductive invariant we'll use to prove P will be:

$$Inv \triangleq x \in \{0, 1\}$$

Now, the formal proof (which can be mechanically checked by TLAPS):

THEOREM $Spec \Rightarrow \Box P$
$\langle 1 \rangle 1.$ $Inv \Rightarrow P$
BY DEF Inv, P
$\langle 1 \rangle 2.$ $Init \Rightarrow Inv$
BY DEF $Inv, Init$
$\langle 1 \rangle 3.$ $Inv \wedge [Next]_x \Rightarrow Inv'$
$\langle 2 \rangle$ SUFFICES ASSUME $Inv, [Next]_x$
PROVE Inv'
$\langle 2 \rangle$ USE DEF $Inv, Init$ Implicitly use these definitions below
$\langle 2 \rangle 1.$ CASE $x' = 1 - x$
OBVIOUS
$\langle 2 \rangle 2.$ CASE UNCHANGED x
OBVIOUS
$\langle 2 \rangle$ QED
BY $\langle 2 \rangle 1, \langle 2 \rangle 2$
$\langle 1 \rangle$ QED
BY $\langle 1 \rangle 1, \langle 1 \rangle 2, \langle 1 \rangle 3, PTL$ DEF $Spec$

Finally, let's consider a short exercise given by Lamport in his note ***Teaching Concurrency***:

> *Consider N processes numbered from 0 through N − 1 in which each process i executes*
>
> `x[i] := 1`
> `y[i] := x[(i - 1) mod N]`
>
> *and stops, where each* `x[i]` *initially equals 0. (The reads and writes of each* `x[i]` *are assumed to be atomic.) This algorithm satisfies the following property: after every process has stopped,* `y[i]` *equals 1 for at least one process i. It is easy to see that the algorithm satisfies this property; the last process i to write* `y[i]` *must set it to 1. But that process doesn't set* `y[i]` *to 1 because it was the last process to write* `y`*. What a process does depends only on the current state, not on what processes wrote before it. The algorithm satisfies this property because it maintains an inductive invariant. Do you know what that invariant is? If not, then you do not completely understand why the algorithm satisfies this property.*

Because this is clearly a code-level algorithm, it is more convenient to write it in PlusCal, a pseudocode-language that is translated to TLA$^+$. The language should be self-explanatory, except for the labels, that specify atomicity: all statements between two consecutive labels are executed atomically:

$$
\begin{array}{l}
\textbf{--algorithm}\ Example\ \{ \\
\quad \textbf{variables}\ x = [i \in 0..N-1 \mapsto 0], \\
\qquad\qquad\qquad\ y = [i \in 0..N-1 \mapsto 0]; \\
\\
\quad \textbf{process}\ (Proc \in 0..N-1)\ \{ \\
\qquad p1 : x[self] := 1; \\
\qquad p2 : y[self] := x[(self-1)\%N]; \\
\quad \} \\
\}
\end{array}
$$

Note that I've chosen to make reading x before writing y in $p2$ atomic.

When writing this PlusCal algorithm in a special comment box, the TLA$^+$ toolbox translates it into the following TLA$^+$ specification:

$$
\begin{array}{l}
\text{VARIABLES}\ x, y, pc \\
vars \triangleq \langle x, y, pc \rangle \\
ProcSet \triangleq 0..N-1 \\
\\
Init \triangleq \land x = [i \in 0..N-1 \mapsto 0] \\
\qquad\ \land y = [i \in 0..N-1 \mapsto 0] \\
\qquad\ \land pc = [self \in ProcSet \mapsto \text{“p1 ”}] \\
\\
p1(self) \triangleq \land pc[self] = \text{“p1 ”} \\
\qquad\qquad\ \land x' = [x\ \text{EXCEPT}\ ![self] = 1] \\
\qquad\qquad\ \land pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“p2 ”}] \\
\qquad\qquad\ \land y' = y \\
\\
p2(self) \triangleq \land pc[self] = \text{“p2 ”} \\
\qquad\qquad\ \land y' = [y\ \text{EXCEPT}\ ![self] = x[(self-1)\%N]] \\
\qquad\qquad\ \land pc' = [pc\ \text{EXCEPT}\ ![self] = \text{“Done ”}] \\
\qquad\qquad\ \land x' = x \\
\\
Proc(self) \triangleq p1(self) \lor p2(self) \\
\\
Next \triangleq \lor \exists self \in 0..N-1 : Proc(self) \\
\qquad\ \lor (\forall self \in ProcSet : pc[self] = \text{“Done ”}) \land \text{UNCHANGED}\ vars \\
\\
Spec \triangleq Init \land [Next]_{vars}
\end{array}
$$

The property we wish to verify is $AtLeastOne1$:

$$
\begin{array}{l}
Done(p) \triangleq pc[p] = \text{“Done ”} \\
AllDone \triangleq \forall p \in ProceSet : Done(p) \\
\\
AtLeastOne1 \triangleq AllDone \Rightarrow \exists p \in ProceSet : y[p] = 1
\end{array}
$$

The challenge is finding an inductive invariant, but the TLC model checker can assist with that. We begin with a guess for Inv. Assuming that our algorithm is written in the normal form, $Spec \triangleq Init \wedge [Next]_v \wedge Fairness$, first we make sure that Inv is truly an invariant by letting the model checker verify that $Spec \Rightarrow \Box Inv$. Then, Inv is an *inductive* invariant iff $Init \Rightarrow Inv$ (also easy to check) *and* $Inv \wedge [Next]_v \Rightarrow \Box Inv$. As $Inv \wedge [Next]_v$ is itself written in normal form, only one where Inv is the initial condition, this, too, can be checked by the model checker (TLC, the model checker included in the TLA+ tools can only check temporal formulas containing actions if they're written in normal form).

In our case, the inductive invariant, Inv is:

$$\begin{aligned} TypeOk \triangleq & \wedge pc \in [ProcSet \to \{\text{“p1 ”}, \text{“p2 ”}, \text{“Done ”}\}] \\ & \wedge x \ \in [ProcSet \to \{0, 1\}] \\ & \wedge y \ \in [ProcSet \to \{0, 1\}] \end{aligned}$$

$$\begin{aligned} Inv \quad \triangleq & \wedge TypeOK \\ & \wedge AtLeastOne1 \\ & \wedge \forall p \in ProcSet : pc[p] \neq \text{“p1 ”} \Rightarrow x[p] = 1 \end{aligned}$$

The secret to how the algorithm ensures that $AtLeastOne1$ holds is in the last conjunct of Inv.

Because TLA allows us to explicitly mention the program's control state in the inductive invariant (e.g., the control point — the program counter — or the state of the call stack), it is more convenient than Floyd-Hoare or Owicki-Gries. The inductive invariant method it is relatively complete with respect to proving invariants (i.e., if the assertions about the action can be proven, so can the invariant). For a more thorough discussion of the completeness of the inductive invariant method and comparison with other approaches see Lamport's ***Computation and State Machines***.

As to the utility and necessity of explicitly mentioning control state, Lamport **writes**:

> *There used to be an incredible reluctance by theoretical computer scientists to mention the control state of a program. When I first described the work … to Albert Meyer, he immediately got hung up on the control predicates. We spent an hour arguing about them — I saying that they were necessary (as was first proved by Susan [Owicki] in her thesis), and he saying that I must be doing something wrong. I had the feeling that I was arguing logical necessity against religious belief, and there's no way logic can overcome religion.*

In practice, however, when specifying complex algorithms — let alone large and complex real-world software systems — formal proofs become very costly and are rarely what you need. Finding an inductive invariant of a complex algorithm is not easy. Using the model checker is likely to be at least an order of magnitude cheaper and give you sufficient confidence (TLC checks not only safety but also liveness properties). It almost always yields a better return on investment. Currently, formal proofs are used mostly by academics either to prove the correctness of a novel general algorithm or to demonstrate the viability of the approach in principle. In industry, formal proofs are used either for very small and very critical modules, or (especially in hardware) to tie together results about the correctness of a composition of modules, each verified with a model checker or some other automatic method. Even if you choose to write a formal proof, you should check your theorem in a model-checker first. As Lamport says, it's much easier to prove something if it's true.