The *Principles* and *Specification* Tracks

1 Introduction

1.1 Concurrent Computation

Concurrent means occurring at the same time. *Concurrency* is the noun form of this adjective; it means the existence of multiple things happening at the same time.

Concurrent computation means computation in which different operations can occur concurrently. These days, most computation is performed in response to real-world actions—perhaps when a user moves a mouse or clicks on a mouse button. Concurrency in the real world means that concurrent computation cannot be avoided. Your computer cannot prevent you from clicking on the mouse button while you are moving the mouse.

Parallel computation is a special kind of concurrent computation in which different parts of a single task are performed concurrently to speed up execution of the task. In principle, parallelism is avoidable because we can perform the separate parts one at a time. It may not be avoidable in practice because without it, executing the task may take too long. However, parallelism is an inherently simpler form of concurrency because we, rather than the external world, control when things happen.

1.2 Modeling Computation

Concurrent computation is computation in which different operations can occur concurrently, but what is computation? A simple answer is: computation is what a computer does. This answer is unsatisfactory for several reasons:

- It's hard to define what a computer is. Is a cell phone a computer? What about an MP3 player?

- These days, computations are often performed by networks of separate computers.

- Computations can be performed by non-physical things—in particular, by programs and algorithms.

A better definition is that computation is what a digital system does, where computers, MP3 players, computer networks, programs, and algorithms are all digital systems. What distinguishes a digital system is that its computation consists of a collection of discrete events.

A pocket calculator is a digital system because its computation consists of discrete events like the pressing of a button and the writing of a number on its display. But are these really discrete events? Changing the number shown on the display requires a few milliseconds, during which time the display changes

continuously from showing its old value to showing its new one. The user of the calculator thinks of it as a single event. The designer of the display probably doesn't. We consider something to be a digital system if we think of its computation as consisting of discrete events. However, instead of saying that we are considering the calculator to be a digital system, we simply say that it *is* one. Moreover, since this hyperbook is about digital systems, I will almost always omit the "digital" and simply write *system* to mean digital system.

What exactly are the discrete events of the pocket calculator system? Is entering the number 3 on the keypad a single event? Or are depressing the 3 and releasing it two separate events? A user of the calculator probably considers entering 3 to be a single event; to the keypad's designer, they are separate events.

This hyperbook is not about physical systems like calculators. It is about *abstract* systems, which are abstractions of digital systems obtained by considering their computations to consist of certain discrete events. The principles we study are principles of abstract systems.

How do we decide what abstraction of a physical system to use? Should entering the number 3 be one event or two? The answer depends on the purpose of the abstraction. The abstraction in which entering a number is a single event is simpler. However, it cannot describe the physical possibility of depressing the 3 and then depressing the 4 before releasing the 3. The keypad engineer cares about this possibility, so the abstraction does not serve her purpose and she needs separate *depress* and *release* events. The user trying to understand how to use the calculator probably doesn't care what happens if two keys are depressed at the same time, so he will prefer an instruction manual that adopts the simpler abstraction.

This kind of abstraction is common to all sciences. Astronomers studying planetary motion often use an abstraction in which a planet is represented as a point mass. However, that abstraction is not satisfactory if tidal effects are important.

Having fewer separate events makes an abstraction simpler; having more events allows it to more accurately represent the actual system. We want to use the simplest abstraction that is accurate enough. Finding the right abstraction is an art, but we will see that there are principles that can guide us.

Having chosen an abstraction of a system, we need to decide how to represent that abstraction. A representation of an abstraction of a system is called a *model* of the system. There are several ways of modeling systems. Some take events to be primitive objects. Others take states to be primitive, where a state is an assignment of values to variables, with an event defined to be a transition from one state to another. Still others take both states and event names as primitives, with an event being a state transition labeled by an event name. There is also one way of modeling systems in which the primitive objects are sets of events. These different kinds of models can be used to express different classes of properties, and we will use more than one of them. However, the one

we take as our standard model, and the one we use most often, is:

> **The Standard Model** An abstract system is described as a collection of behaviors, each representing a possible execution of the system, where a behavior is a sequence of states and a state is an assignment of values to variables.

In this model, an event, also called a *step*, is the transition from one state to the next in a behavior. I find the standard model to be the simplest one that scales well to descriptions of complex systems.

We model abstractions of systems. By a *system model* or a *model of a system*, I mean a model of an abstraction of a (digital) system.

1.3   Specification

A *specification* is a description of a system model. A *formal* specification is one that is written in a precisely defined language. I will use the term *system specification* (or *specification of a system*) to mean a specification of a system model.

A system specification is a specification of a model of an abstraction of a system. It is quite removed from an actual system. Why should we write such a specification?

A specification is like a blueprint. A blueprint is far removed from a building. It is a sheet of paper with writing on it, while a building is made of steel and concrete. There is no need to explain why we draw blueprints of buildings. However, it's worth pointing out that a blueprint is useful in large part because it is so far removed from the building it is describing. If you want to know how many square feet of office space the building has, it is easier to use a blueprint than to measure the building. It is very much easier if the blueprint was drawn with a computer program that can automatically calculate such things.

No one constructs a large building without first drawing blueprints of it. We should not build a complex system without first specifying it. People will give many reasons why writing a specification of a system is a waste of time:

- You can't automatically generate code or circuit diagrams from the specification.

- You (usually) can't verify that the code or circuit diagrams correctly implement the specification.

- While building the system, you can discover problems that require changing what you want the system to do. This leads to the specification not describing the actual system.

You can find the answers to such arguments by translating them into the corresponding ones for not drawing blueprints.

Blueprints are most useful when drawn before the building is constructed, so they can guide its construction. However, they are sometimes drawn afterwards—for example, before remodeling an old building whose blueprints have been lost. System specifications are also most useful before the system is built. However, they are also written afterwards to understand what the system does—perhaps to look for errors or because the system needs to be modified.

A formal specification is like a detailed blueprint; an informal specification is like a rough design sketch. A sketch may suffice for a small construction project such as adding a skylight or a door to a house; an informal specification may suffice for a simple system model. The main advantage of writing a formal specification is that you can apply tools to check it for errors. This hyperbook teaches you how to write formal specifications and how to check them. Learning to write formal specifications will help you to write informal ones.

1.4   Systems and Languages

A formal specification must be written in a precisely defined language. What language or languages should we use?

A common belief is that a system specification is most useful if written in a language that resembles the one in which the system is to be implemented. If we're specifying a program, the specification language should look like a programming language. By this reasoning, if we construct a building out of bricks, the blueprints should be made of brick.

A specification language is for describing models of abstractions of digital systems. Most scientists and engineers have settled on a common informal language for describing models of abstractions of non-digital systems: the language of mathematics. Mathematics is the simplest and most expressive language I know for describing digital systems as well.

Although mathematics is simple, the education of programmers and computer scientists (at least in the United States) has made them afraid of it. Fortunately, the math that we need for writing specifications is quite elementary. I learned most of it in high school; you should have learned most of it by the end of your first or second year of university. What you need to understand are the elementary concepts of sets, functions, and simple logic. You should not only understand them, but they should be as natural to you as simple arithmetic. If you are not already comfortable with these concepts, I hope that you will be after reading and writing specifications.

Although mathematics is simple, we are fallible. It's easy to make a mistake when writing mathematical formulas. It is almost as hard to get a formula right the first time as it is to write a program that works the first time you run it. For them to be checked with tools, our mathematical specifications must be formal ones. There is no commonly accepted formal language for writing mathematics, so I had to design my own specification language: TLA$^+$.

The TLA⁺ language has some [notations and concepts that are not ordinary math](#), but you needn't worry about them now. You'll quickly get used to the notations, and the new concepts are either "hidden beneath the covers", or else they are used mainly for advanced applications.

Although mathematics is simple and elegant, it has two disadvantages:

- For many algorithms, informal specifications written in pseudo-code are simpler than ones written in mathematics.

- Most people are not used to reading mathematical specifications of systems; they would prefer specifications that look more like programs.

PlusCal is a language for writing formal specifications of algorithms. It resembles a very simple programming language, except that any TLA⁺ expression can be used as an expression in a PlusCal algorithm. This makes PlusCal infinitely more expressive than any programming language. An algorithm written in PlusCal is translated (compiled) into a TLA⁺ specification that can be checked with the TLA⁺ tools.

PlusCal is more convenient than TLA⁺ for describing the flow of control in an algorithm. This generally makes it better for specifying sequential algorithms and shared-memory multiprocess algorithms. Control flow within a process is usually not important in specifications of distributed algorithms, and the greater expressiveness of TLA⁺ makes it better for these algorithms. However, TLA⁺ is usually not much better, and the PlusCal version may be preferable for people less comfortable with mathematics. Most of the algorithms in this hyperbook are written in PlusCal.

Reasoning means mathematics, so if you want to prove something about a model of a system, you should use a TLA⁺ specification. PlusCal was designed so the TLA⁺ translation of an algorithm is straightforward and easy to understand. Reasoning about the translation is quite practical.

2 The One-Bit Clock

Our first example is a clock. We consider the simplest possible clock: one that alternately shows the "times" 0 and 1. Such a clock controls the computer on which you are reading this, with its times being displayed as the voltage on a wire. A real clock should tick at an approximately constant rate. There is a lot to explain before we can specify that requirement, so we are going to ignore it. This leaves a very simple computing device that just alternates between two states: the state in which the clock displays 0 and the state in which it displays 1.

This may seem a strange example to choose because it has no concurrency. The clock does only one thing at a time. A system can do any number of things at a time. *One* is a simple special case of *any number*, and it's a good place to begin. Learning to specify sequential systems in TLA$^+$ teaches most of what you need to know to specify concurrent systems.

2.1 The Clock's Behaviors

We use the standard model to represent the clock. This means that a possible execution of the clock is represented by a behavior, which is a sequence of states, and a state is an assignment of values to variables. We model the clock with a single variable b that represents the clock "face", where the assignment of 0 to b represents the clock displaying 0, and the assignment of 1 to b represents its displaying 1. We describe the state that assigns the value 0 to b by the formula $b = 0$, and similarly for $b = 1$.

If we start the clock displaying 0, then we can pictorially represent its behavior as:

$$b = 0 \quad \to \quad b = 1 \quad \to \quad b = 0 \quad \to \quad b = 1 \quad \to \quad \cdots$$

where "$\cdots$" means that the clock goes on forever the same way. Real clocks eventually stop; the best we can expect is that they keep running for long enough. However, it's more convenient to consider an ideal clock that never stops, rather than having to decide for how long we should require it to run. So, we describe a clock that runs forever.

We could also let the clock start displaying 1, in which case its behavior is

$$b = 1 \quad \to \quad b = 0 \quad \to \quad b = 1 \quad \to \quad b = 0 \quad \to \quad \cdots$$

These two are the only possible behaviors of the one-bit clock.

Remember that, although I have been calling them behaviors of the clock, what I have really described are the behaviors in the standard model of an abstraction of a real clock. The display of a clock moves continuously from one value to

the next. In a digital clock, the transition may be too fast for us to see the intermediate values; but they are there. We are specifying an abstraction of the clock in which these continuous changes are represented by discrete steps (state changes).

2.2 Describing the Behaviors

To describe a computing device, we must describe all its possible behaviors. I was able to list all the possible behaviors of the one-bit clock, but that isn't feasible for any but the simplest computing devices. Even displaying a single behavior of a complex device would be hard, and most computing devices have too many behaviors to list—often, infinitely many behaviors.

If we look beyond their syntax, we find that practical languages for describing computing devices specify two things:

- The possible initial states.

- The possible steps. (Remember that a step is a transition from one state to the next.)

For example, here's how the one-bit clock might be described in a (nonexistent) programming language.

> **variable** b: 0, 1;
>
> **while (true)** { **if** $(b = 0)$ b := 1 **else** b := 0; }

The first line says that the possible initial states are $b = 0$ and $b = 1$. The second line says that if b equals 0, then in the next state it equals 1; and if it equals 1, then in the next state it equals 0.

Instead of inventing a whole new language for describing initial states and possible next states, we will do it with mathematics. We do this using the Boolean operators $\wedge$ and $\vee$. If you are not as familiar with these operators of simple logic as you are with the operators $+$ and $-$ of arithmetic, you should detour to a discussion of logic.$^{\square}$.

Describing the initial states is simple; we just assert that the initial value of b is 0 or 1. This assertion is expressed by the formula:

$$(b = 0) \ \vee \ (b = 1)$$

We call this formula the *initial predicate*.

To describe the possible steps, we have to write a mathematical formula relating the values of b in two states: the first state of the step and its next state. We do this by letting b mean the value of b in the first state, and b' mean its value in the next state. There are two possible steps: one with $b = 0$ and $b' = 1$, and the other with $b = 1$ and $b' = 0$. Thus, all possible steps are described by this formula:

$$((b = 0) \ \wedge \ (b' = 1)) \ \vee \ ((b = 1) \ \wedge \ (b' = 0))$$

Even this tiny formula is a little hard to read because of all the parentheses. For larger formulas with conjunctions and disjunctions, it can get almost impossible to keep track of the parentheses. TLA$^+$ allows us to write conjunctions and disjunctions as lists of formulas bulleted by $\wedge$ or $\vee$. We can therefore also write this formula as

$$\begin{array}{lll} \vee\ (b = 0)\ \wedge\ (b' = 1) & \quad \text{or} \quad & \vee\ \wedge\ b = 0 \\ \vee\ (b = 1)\ \wedge\ (b' = 0) & & \quad\ \wedge\ b' = 1 \\ & & \vee\ \wedge\ b = 1 \\ & & \quad\ \wedge\ b' = 0 \end{array}$$

However it is written, we usually call this formula the *next-state action* or sometimes the *next-state relation*.

2.3 Writing the Specification

Let's now turn the initial predicate and next-state action into a TLA$^+$ specification. Open a new spec in the TLA$^+$ Toolbox. Name the specification and its root module *OneBitClock*. This creates a new module file named `OneBitClock.tla` and opens an editor on it.

The newly created module looks something like this in the editor:

```
------------------ MODULE OneBitClock -------------------


=============================================================
\* Modification History
\* Created Mon Dec 13 09:57:04 PST 2010 by jones
```

The first line is the *module opening*; the last line is the *module closing*. All text before the opening and after the closing is not part of the module and is ignored. Each sequence of `-` characters in the opening and the sequence of `=` characters in the closing can be of any length greater than 3. The opening and closing are printed as follows:

$$\boxed{\qquad\qquad\qquad \text{MODULE } \textit{OneBitClock} \qquad\qquad\qquad}$$

We now assign names to our initial predicate and next-state action. I have traditionally called them *Init* and *Next*. However, we will be defining some alternative initial predicates and next-state relations, so let's call these *Init*1 and *Next*1. These formulas are defined as follows.

$$Init1 \ \triangleq\ (b = 0) \vee (b = 1)$$

$$Next1 \ \triangleq\ \vee\ \wedge\ b = 0$$

$$\wedge\ b' = 1$$
$$\vee\ \wedge\ b = 1$$
$$\wedge\ b' = 0$$

These two TLA$^+$ statements define *Init1* and *Next1* to be the two formulas. Thus, anywhere in the spec following the definition of *Init1*, typing *Init1* is completely equivalent to typing $((b = 0) \vee (b = 1))$. The symbol $\triangleq$ (typed ==) is read *is defined to equal*.

Now save the module, which should cause the Toolbox to parse the module. (If it doesn't, go to the TLA+ Parser Preferences menu.) The parser will report six errors, all complaining that b is an unknown operator. Clicking on each error message in the Parsing Errors view highlights the location of the error—in this case, the location of the particular occurrence of b that it is complaining about.

Every symbol that appears in the module must either be a primitive TLA$^+$ operator or else defined or declared before its first use. We must declare b to be a variable, which we do by inserting the following declaration at the beginning of the module, before the definitions of *Init1*.

VARIABLE b VARIABLE b

Saving will make the errors go away.

———

When talking about the *specification* of the one-bit clock, we can mean one of two things:

- The complete module.

- The initial predicate and next-state relation.

It is usually clear from the context which is meant. To avoid confusion, we can talk about the module rather than the specification when we mean the first. We use the term *behavior specification* to mean the second.

2.4   The Pretty-Printed Version of Your Spec

In addition to the ASCII version of the module that you edit, the Toolbox can display a "pretty-printed" version. This requires the `pdflatex` program to be installed on your computer. Information on doing that and on configuring the Toolbox's pretty-printing options can be found in the relevant Toolbox help page.

To produce a pretty-printed version of the module, click on the File menu and choose Produce PDF Version. The pretty-printed version will be displayed in a separate window within the Toolbox, with TLA$^+$ expressions shown approximately the way they are printed in this hyperbook. You can switch between the ASCII and pretty-printed versions by clicking either the TLA Module or PDF

Viewer tab in the top-left corner of the module's window. Editing the ASCII version does not automatically change the pretty-printed version. You need to run the File / Produce PDF Version command again to update it.

The pretty-printed version is produced in a file `OneBitClock.pdf` that the Toolbox puts in the same directory as the module file `OneBitClock.tla`. You can print that file to get a paper version.

2.5   Checking the Specification

Let's now get the TLC model checker to check this specification. Create a new model. This opens a model editor on the model. That editor has three pages; the model is opened to the Model Overview page.

Enter *Init*1 and *Next*1 in the appropriate fields as the initial predicate and next-state relation, and run TLC. TLC runs for a couple of seconds and stops, reporting no errors. This means that the specification is sensible. More precisely, it means that our specification completely determines a collection of behaviors.

––––––––

Let's change the specification so it doesn't determine a collection of behaviors. Go to the module editor (by clicking on its tab) and modify the definition of *Next*1 by replacing `/\ b' = 0` with `/\ b' = "xyz"`. The second disjunct allows a step starting with $b = 0$ to set b (change its value) to the string□ "xyz". Save the module, return to the model editor, and run TLC again. This time it reports the error:

```
Attempted to check equality of string "xyz" with non-string:  0
```

The TLC Errors window also shows:

You can resize the fields of the TLC Errors view.

This describes the following error trace:

$$b = 1 \quad \rightarrow \quad b = \text{"xyz"}$$

The trace is the beginning of a behavior that TLC was constructing when it encountered an error. The light-red background for the value "xyz" of b indicates that it is different from the value of b in the previous state. Double click on this line of the error trace:

This raises the module editor, showing in part:

```
6  Next1 == \/ /\ b = 0
7           /\ b' = 1
8       \/ /\ b = 1
9           /\ b' = "xyz"
```

The highlighted portion is the disjunct of the next-state action *Next1* that permits the step $b = 1 \rightarrow b = \text{"xyz"}$.

To calculate the possible next states from the state with $b = \text{"xyz"}$, TLC had to compute the value of the formula $\text{"xyz"} = 0$. (The rest of the error message tells you that it was computing that formula in order to evaluate the subformula $b = 0$ of the definition of *Next1*.) TLC couldn't do that because the semantics of TLA$^+$ do not determine whether or not a string is equal to a number. It could therefore not determine if the formula $\text{"xyz"} = 0$ equals TRUE or FALSE, so it reported an error.

Why shouldn't "xyz" be unequal to 0?

Restore the original definition of *Next1* by replacing "xyz" with 0 and save the module. Go back to the model editor and run TLC again. It should once again find no error.

═══════

In the Statistics section of the Model Checking Results page, the State space progress table tells you that TLC found 2 distinct states. The diameter of 1 means that 1 is the largest number of steps (transitions from one state to the next) that an execution of the one-bit clock can take before it repeats a state.

The one-bit clock is so simple there isn't much to check. But there is one property that we can and should check of just about any spec: that it is "type correct". Type correctness of a TLA$^+$ specification means that in every state of every behavior allowed by the spec, the value of each variable is in the set of values that we expect it to have. For the one-bit clock, we expect the value of b always to be either 0 or 1. This means that we expect the formula $b \in \{0, 1\}$ to be true in every state of every behavior of b. If you are the least bit unsure of what this formula means, detour to an introduction to sets$^\square$.

A formula that is true in all states of all behaviors allowed by a spec is called an *invariant* of the spec. Go to the Invariants subsection of the What to Check section of the model editor's Model Overview page. Open that subsection (by clicking on the +), click on Add, and enter the following formula:

Use the tabs at the top of the model editor view to select the page.

$$b \in \{0, 1\} \qquad\qquad \texttt{b \in \{0, 1\}}$$

(Note that $\in$ is typed \in.) Click on Finish, and then run TLC again on the model. TLC should find no errors, indicating that this formula is an invariant of the spec.

Because TLA$^+$ has no types, it has no type declarations. As this spec shows, there is no need for type declarations. We don't need to declare that b is of type $\{0, 1\}$ because that's implied by the specification. However, the reader of the spec doesn't discover that until after she has read the definitions of the initial predicate and next-state action. In most real specifications, it's hard to

understand those definitions without knowing what the set of possible values of each variable is. It's a good idea to give the reader that information by defining the type-correctness invariant in the spec, right after the declaration of the variables. So, let's add the following definition to our spec, right after the declaration of b.

$$TypeOK \;\triangleq\; b \in \{0,\,1\} \qquad\qquad \texttt{TypeOK == b \in \{0,1\}}$$

Save the spec and let's tidy up the model by using *TypeOK* rather than $b \in \{0,\,1\}$ as the invariant. Go to the model editor's Model Overview page, select the invariant you just entered by clicking on it and hit Edit (or simply double-click on the invariant), and replace the formula by TypeOK. Click on Finish and run TLC to check that you haven't made a mistake.

2.6 Computing the Behaviors from the Specification

TLC checked that *TypeOK* is an invariant of the specification of the one-bit clock, meaning that it is true in all states of all behaviors satisfying the specification. TLC did this by computing all possible behaviors that satisfy the initial predicate *Init1* and the next-state action *Next1*. To understand how it does this, let's see how we can do it.

We begin by computing one possible behavior. A behavior is a sequence of states. To satisfy the spec, the behavior's first state must satisfy the initial predicate *Init1*. A state is an assignment of values to all the spec's variables, and this spec has only the single variable b. So to determine a possible initial state, we must find an assignment of values to the variable b that satisfy *Init1*. Since *Init1* is defined to equal

$$(b = 0) \vee (b = 1)$$

there are obviously two such assignments: letting b equal 0 or letting it equal 1. To construct one possible behavior satisfying the spec, let's arbitrarily choose the starting state in which b equals 1. As before, we write that state as the formula $b = 1$.

We next find a possible second state of the behavior. For a behavior to satisfy the spec, every pair of successive states must satisfy the next-state action *Next1*, where the values of the unprimed variables are the values assigned to them by the first state of the pair and the values of the primed variables are the values assigned to them by the second state of the pair. The first state of our behavior is $b = 1$. To obtain the second state, we need to find a value for b' that satisfies *Next1* when b has the value 1. We then let b equal that value in the second state. To find this value, we substitute 1 for b in *Next1* and simplify the formula.

Recall that *Next1* is defined to equal

$$\lor \; \land \; b = 0$$
$$\qquad \land \; b' = 1$$
$$\lor \; \land \; b = 1$$
$$\qquad \land \; b' = 0$$

We substitute 1 for b and simplify as follows.

$$\lor \; \land \; 1 = 0 \qquad \text{the formula obtained by substituting 1 for } b \text{ in } \textit{Next1}.$$
$$\qquad \land \; b' = 1$$
$$\lor \; \land \; 1 = 1$$
$$\qquad \land \; b' = 0$$

$$= \; \lor \; \land \; \text{FALSE} \qquad \text{because } (0 = 1) = \text{FALSE} \text{ and } (1 = 1) = \text{TRUE}$$
$$\qquad\quad \land \; b' = 1$$
$$\qquad \lor \; \land \; \text{TRUE}$$
$$\qquad\quad \land \; b' = 0$$

$$= \; \lor \; \text{FALSE} \qquad \text{because } \text{FALSE} \land F = \text{FALSE} \text{ and } \text{TRUE} \land F = F$$
$$\quad\; \lor \; b' = 0 \qquad \text{for any truth value } F$$

$$= \; b' = 0 \qquad \text{because } \text{FALSE} \lor F = F \text{ for any truth value } F.$$

This computation shows that if we substitute 1 for b in *Next1*, then the only value we can then substitute for b' that makes *Next1* true is 0. Hence, the second state of our behavior can only be $b = 0$, and our behavior starts with

$$b = 1 \;\; \rightarrow \;\; b = 0$$

To find the third state of our behavior, we substitute 0 for b in *Next1* and find a value for b' that makes *Next1* true. It should be clear that the same type of calculation we just did shows that the only possible value for b' that makes *Next1* true is 1. (If it's not clear, go ahead and do the calculation.) The first three states of our behavior therefore must be

$$b = 1 \;\; \rightarrow \;\; b = 0 \;\; \rightarrow \;\; b = 1$$

We could continue our calculations to find the fourth state of the behavior, but we don't have to. We've already seen that the only possible state that can follow $b = 1$ is $b = 0$. We can deduce that we must obtain the infinite behavior

$$b = 1 \;\; \rightarrow \;\; b = 0 \;\; \rightarrow \;\; b = 1 \;\; \rightarrow \;\; b = 0 \;\; \rightarrow \;\; \cdots$$

To find all possible behaviors, recall that the only other possible starting state is $b = 0$. From the calculations we've already done, we know that the only state that can follow $b = 0$ is $b = 1$. We therefore see that the only other possible behavior is

$$b = 0 \;\; \rightarrow \;\; b = 1 \;\; \rightarrow \;\; b = 0 \;\; \rightarrow \;\; b = 1 \;\; \rightarrow \;\; \cdots$$

This example shows how we can compute all possible behaviors allowed by a specification. We construct as follows a directed graph $\mathcal{G}$, called the *state graph*, whose nodes are states:

1. We start by setting $\mathcal{G}$ to the set of all possible initial states of behaviors, which we find by computing all possible assignments of values to variables that make the initial predicate true.

2. For every state s in $\mathcal{G}$, we compute as follows all possible states t such that $s \rightarrow t$ can be a step in a behavior. We substitute the values assigned to variables by s for the unprimed variables in the next-state action, and then compute all possible assignments of values to the primed variables that make the next-state action true.

3. For every state t found in step 2: (i) we add t to $\mathcal{G}$ if it is not already in $\mathcal{G}$, and (ii) we draw an edge from s to t.

4. We repeat steps 2 and 3 until no new states or edges can be added to $\mathcal{G}$.

If and when this process terminates, the nodes of $\mathcal{G}$ consist of all the reachable states of the specifications—that is, all states that occur in some behavior satisfying the specification. Every behavior satisfying the specification can be found by starting in an initial state (found in step 1) and following a (possibly infinite) path in $\mathcal{G}$.

This procedure is used by TLC to compute all possible behaviors. The *State space progress* table in the Statistics section of the Model Checking Results page gives the following information about the graph $\mathcal{G}$ that it is constructing.

Diameter The number of states in the longest path of $\mathcal{G}$ in which no state appears twice.

States Found The total number of (not necessarily distinct) states it examined in step 1 or as successor states t in step 2.

Distinct States The number of states that form the set of nodes of $\mathcal{G}$.

Queue Size The number of states s in $\mathcal{G}$ for which step 2 has not yet been performed.

Of course, if the specification has an infinite number of reachable states, this procedure will continue until $\mathcal{G}$ becomes so large that TLC runs out of space. However, this could take many years because TLC keeps $\mathcal{G}$ and its queue of unexamined states on disk when there is not enough room for them in memory.

Although TLC computes the behaviors that satisfy a specification the same way we do, it's not nearly as smart as we are. For example, writing $1 = b$ instead of $b = 1$ in the initial predicate would make no difference to us. See how TLC

reacts by making this change to the definition of *Init1* in module *OneBitClock* and running TLC on the model you created. You will find that it produces the following error report:

```
In evaluation, the identifier b is either undefined or not an operator.
line 6, col 22 to line 6, col 22 of module OneBitClock.
The error occurred when TLC was evaluating the nested
expressions at the following positions:
0.  Line 6, column 22 to line 6, column 22 in OneBitClock
```

The underlined location indicators are links. (They may not actually be under-lined in the Toolbox.) Clicking on either of them jumps to and highlights the b in $1 = b$.

TLC tries to find all possible initial states from the initial predicate in a very simple-minded way. It examines the predicate in a linear fashion to try to find all possible assignments of values to the variables. When it encounters an occurrence of a variable v whose value it has not yet determined, that occurrence must very obviously determine the value of v. This means that the occurrence must be in a formula $v = e$ or $v \in e$ for some expression e that does not contain v. For example, when TLC evaluated the initial predicate

$$(b = 0) \vee (1 = b)$$

it first saw that it was a disjunction, so it examined the two disjuncts separately. The first disjunct, $b = 0$, has the right form to determine the value of b—that is, it has the form $v = e$ where v is the variable b and e is the expression 0. However, when examining the disjunct $1 = b$, it first encountered the variable b in an expression that did not have the right form. It therefore reported that occurrence of b as an error. You can check that TLC has no problem with the equivalent initial predicate

$$(b = 0) \vee ((b = 1) \wedge (1 = b))$$

because, when it encounters the expression $1 = b$, it has already determined the value of b.

Question 2.1 What happens if you change the initial predicate to   ANSWER

$$(b = 0) \vee ((b = 1) \wedge (2 = b))$$

and run TLC.

These same remarks apply to the way TLC determines the possible assignments to the primed variables from the next-state action when performing step 2 of the procedure above. The first time TLC encounters a primed variable v' whose value it has not yet determined, that occurrence must be in a formula $v' = e$ or $v' \in e$ for some expression e not containing v'.

2.7 Other Ways of Writing the Behavior Specification

If you are not intimately acquainted with the propositional-logic operators $\Rightarrow$ (implication), $\equiv$ (equivalence), and $\neg$ (negation), detour here.$^\square$

The astute reader will have noticed that the two formulas $Init1$ and $TypeOK$, which equal $(b = 0) \vee (b = 1)$ and $b \in \{0, 1\}$, respectively, both assert that b equals either 0 or 1. In other words, these two formulas are equivalent—meaning that the following formula equals TRUE for any value of b:

$$((b = 0) \vee (b = 1)) \equiv (b \in \{0, 1\})$$

The two formulas can be used interchangeably. To test this, return to the Toolbox and select the Model Overview page of the model editor. Replace $Init1$ by $TypeOK$ in the Init field and run TLC again. You should find that nothing has changed.

There are a number of different ways to write the next-state action. This action should assert that b' equals 1 if b equals 0, and equals 0 if b equals 1. Since the value of b is equal to either 0 or 1 in every state of the behavior, an equivalent way to say this is that b' equals 1 if b equals 0, else it equals 0. This is expressed by the formula $Next2$, that we define as follows.

$$Next2 \;\;\triangleq\;\; b' = \text{IF } b = 0 \text{ THEN } 1 \text{ ELSE } 0 \qquad\qquad \texttt{Next2 == b' = IF b = 0 THEN 1 ELSE 0}$$

The meaning of the IF ... THEN ... ELSE construct should be evident.

Unlike $Init1$ and $TypeOK$, the two formulas $Next1$ and $Next2$ are not equivalent. However, they are equivalent if b equals 0 or 1. More precisely, the following formula equals TRUE for all values of b:

$$TypeOK \;\Rightarrow\; (Next1 \equiv Next2)$$

When used with $Init1$ as the initial predicate, both next-state actions yield specifications for which each state of each behavior satisfies $TypeOK$. Hence, the truth of this formula implies that the two specs are equivalent—meaning that they have the same set of allowed behaviors. Test this by copying and pasting the definition of $Next2$ into the module (anywhere after the declaration of b), saving the module, replacing $Next1$ by $Next2$ in the Next field of the model, and running TLC again.

The method of writing the next-state action that I find most elegant is to use the modulus operator $\%^\square$, where $a \% b$ is the remainder when a is divided by b. Since $0 \% 2 = 0$, $1 \% 2 = 1$, and $2 \% 2 = 0$, it's easy to check that, if b equals 0 or 1, then $Next1$ and $Next2$ are equivalent to the following formula.

$$Next3 \;\;\triangleq\;\; b' = (b + 1) \% 2 \qquad\qquad \texttt{Next3 == b' = (b + 1) \% 2}$$

Add this definition to the module and save the module. This will generate a parsing error, informing you that the operator $\%$ is not defined. The usual

arithmetic operators, including $+$ and $-$, are not built-in operators of TLA$^+$. Instead, they must be imported from one of the standard TLA$^+$ arithmetic modules, using an EXTENDS statement. You will usually want to import the *Integers* module, which you do with the following statement:

EXTENDS *Integers*   `EXTENDS Integers`

Add this statement to the beginning of the module and save the module. Open the model editor's Model Overview page, replace the next-state action *Next2* with *Next3*, and run TLC to check this specification.

════════

Mathematics provides many different ways of expressing the same thing. There are an infinite number of formulas equivalent to any given formula. For example, here's a formula that's equivalent to *Next2*.

IF $b = 0$ THEN $b' = 1$
ELSE $b' = 0$

As *Next1* and *Next2* show, even two next-state actions that are not equivalent can yield equivalent specifications—that is, specifications describing the same sets of behaviors.

Question 2.2 Use the propositional operators $\Rightarrow$ and $\wedge$ to write a next-state action that yields another equivalent specification of the one-bit clock. How many other next-state actions can you find that also produce equivalent specifications?   ANSWER

Question 2.3 Can inequivalent initial predicates produce equivalent specifications?   ANSWER

2.8   Specifying the Clock in PlusCal

We now specify the 1-bit clock as a PlusCal algorithm, which means that we start learning the PlusCal language. If at any point you want to jump ahead, you can read the PlusCal language manual.

In the Toolbox, open a new spec and name the specification and its root module *PCalOneBitClock*. The algorithm is written inside a multi-line comment, which is begun by (* and ended by *). The easy way to create such a comment is to put the cursor at the left margin and type control+o control+s. (You can also right-click and select Start Boxed Comment.) Your file will now look about like this.

```
----------------------- MODULE PCalOneBitClock -----------------------

(****************************************************************************
```

```
                ********************************************************************)
                =================================================================
```

We need to choose an arbitrary name for the algorithm. Let's call it *Clock*. We start by typing this inside the comment:

--algorithm *Clock* { }	`--algorithm Clock {` `}`

The `--` in the token **--algorithm** has no significance; it's just a meaningless piece of required syntax that you're otherwise unlikely to put in a comment.

   The body of the algorithm appears between the curly braces { }. It begins by declaring the variable b and specifying its set of possible initial values

variable $b \in \{0, 1\}$;	`variable b \in {0, 1};`

Next comes the executed code, enclosed in curly braces.

{ **while** (TRUE) { **if** $(b = 0)$ $b := 1$ **else** $b := 0$ } }	ASCII version of the complete algorithm.

You should be able to figure out the meaning of this PlusCal code because it looks very much like code written in C or a language like Java that uses C's syntax. The major difference is that in PlusCal, the equality relation is written = instead of `==`, and assignment is written `:=` instead of `=`. (You can make it look more like C by adding semi-colons after the two assignments.)

Why doesn't PlusCal use = for assignment?

   Save the module. Now call the translator by selecting the File menu's Translate PlusCal Algorithm option or by typing control+t. The translator will insert the algorithm's TLA⁺ translation after the end of the comment containing the algorithm, between the two comment lines:

```
\* BEGIN TRANSLATION   and   \* END TRANSLATION
```

If the file already contains these two comment lines, the translation will be put between them, replacing anything that's already there.

   The important parts of the translation are the declaration of the variable b and the definitions of the initial predicate *Init* and the next-state action *Next*. Those two definitions are the following

$$Init \;\triangleq\; b \in \{0, 1\}$$

$$Next \;\triangleq\; \text{IF } b = 0 \text{ THEN } b' = 1$$
$$\text{ELSE } \quad b' = 0$$

except that the translator formats them differently, inserting a comment and some unnecessary $\wedge$ operators at the beginning of formulas. (A bulleted list of conjuncts can consist of just one conjunct.)

We have seen above that this definition of *Init* is equivalent to the definition of *Init*1 in module *OneBitClock*. We have seen the definition of *Next* above too, where we observed that it is equivalent to the definition of *Next*2 in the *OneBitClock* module.

The translation also produces definitions of the symbols *var* and *Spec*. You should ignore them for now.

As you have probably guessed, if we replace the **if / else** statement in the PlusCal code with the statement $b := (b + 1) \% 2$, the translation will define *Next* to be the formula *Next*3 we defined above. Try it. As before, the Toolbox will complain that $\%$ is undefined. You have to add an EXTENDS *Integers* statement to the beginning of the module.
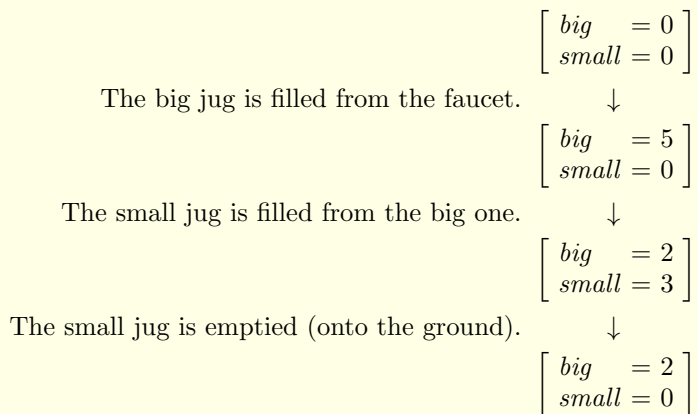
3 The Die Hard Problem

In the movie *Die Hard 3*, the heroes must solve the problem of obtaining exactly 4 gallons of water using a 5 gallon jug, a 3 gallon jug, and a water faucet. We now apply TLA$^+$ and the TLC model checker to solve this problem.

3.1 Representing the Problem in TLA$^+$

The first step in solving the problem is to model the physical system of heroes, jugs, and faucet mathematically as a discrete system. The only relevant state of the hero/jug/faucet system is the amount of water in the two jugs. So, we model the system with two variables, *big* and *small*, whose values represent the number of gallons of water in the two jugs. After choosing the variables, a good way to figure out how to write a specification is to write down the first few states of a possible behavior of the system. Initially, the jugs are empty, so *big* and *small* both equal 0. Here's one possible beginning of a behavior. (Remember that a state is an assignment of values to the variables, in this case *big* and *small*.)

$$\begin{bmatrix} big & = 0 \\ small & = 0 \end{bmatrix}$$

The big jug is filled from the faucet. $\quad \downarrow$

$$\begin{bmatrix} big & = 5 \\ small & = 0 \end{bmatrix}$$

The small jug is filled from the big one. $\quad \downarrow$

$$\begin{bmatrix} big & = 2 \\ small & = 3 \end{bmatrix}$$

The small jug is emptied (onto the ground). $\quad \downarrow$

$$\begin{bmatrix} big & = 2 \\ small & = 0 \end{bmatrix}$$

A little thought reveals that there are three kinds of steps in a behavior:

- Filling a jug.

- Emptying a jug.

- Pouring from one jug to the other. There are two cases:

    - This empties the first jug.

    - This fills the second jug, possibly leaving water in the first jug.

We can now write the specification. Let's open a new specification named *DieHard* in the Toolbox. Since the spec will require arithmetic operations, it begins with:

EXTENDS *Integers*

```
EXTENDS Integers
```

We declare the variables and write the initial predicate, which we give the conventional name *Init*.

VARIABLES *big*, *small*

$Init \triangleq \land big = 0$
$\qquad\qquad \land small = 0$

```
VARIABLES big, small
Init == /\ big = 0
        /\ small = 0
```

Each of the three possible kinds of steps has two possibilities—one for each jug (each first jug for the third type). This suggests writing the next state action as the disjunction of six formulas, each allowing one of these six possible kinds of step. We can therefore define the next-state action, which by convention is called *Next*, as follows:

$Next \triangleq \lor FillSmall$
$\qquad\quad \lor FillBig$
$\qquad\quad \lor EmptySmall$
$\qquad\quad \lor EmptyBig$
$\qquad\quad \lor SmallToBig$
$\qquad\quad \lor BigToSmall$

```
Next == \/ FillSmall
        \/ FillBig
        \/ EmptySmall
        \/ EmptyBig
        \/ SmallToBig
        \/ BigToSmall
```

The definitions of the six formulas *FillSmall*, ..., *BigToSmall*, which often called *subactions* of the next-state action, must precede the definition of *Next* in the module. (In TLA$^+$, a symbol must be defined or declared before it can be used.) Let's now define them.

Most programmers would expect the definition of *FillSmall* to be

$FillSmall \triangleq small' = 3$

This formula is certainly satisfied by a step like

$$\begin{bmatrix} big & = 2 \\ small = 1 \end{bmatrix} \rightarrow \begin{bmatrix} big & = 2 \\ small = 3 \end{bmatrix}$$

However, the formula is also satisfied by this step

$$\begin{bmatrix} big & = 2 \\ small = 1 \end{bmatrix} \rightarrow \begin{bmatrix} big & = \sqrt{42} \\ small = & 3 \end{bmatrix}$$

because substituting

$$big \leftarrow 2, \quad small \leftarrow 1, \quad big' \leftarrow \sqrt{42}, \quad small' \leftarrow 3$$

in the formula produces the true formula $3 = 3$. Since a step that fills the small jug should leave the contents of the big jug unchanged, the subaction *FillSmall* must assert that big' equals big. With this observation, the definitions of the first four subactions are obvious:

$$FillSmall \triangleq \wedge small' = 3$$
$$\wedge big' = big$$

```
FillSmall == /\ small' = 3
             /\ big' = big
```

$$FillBig \triangleq \wedge big' = 5$$
$$\wedge small' = small$$

```
FillBig == /\ big' = 5
           /\ small' = small
```

$$EmptySmall \triangleq \wedge small' = 0$$
$$\wedge big' = big$$

```
EmptySmall == /\ small' = 0
              /\ big' = big
```

$$EmptyBig \triangleq \wedge big' = 0$$
$$\wedge small' = small$$

```
EmptyBig == /\ big' = 0
            /\ small' = small
```

The definitions of the last two, *SmallToBig* and *BigToSmall*, are trickier because each has two cases. Let's consider *SmallToBig*. We can express the two possibilities as the disjunction of two formulas:

$$SmallToBig \triangleq \vee \wedge big + small > 5$$
$$\wedge big' = 5$$
$$\wedge small' = small - (5 - big)$$
$$\vee \wedge big + small \leq 5$$
$$\wedge big' = big + small$$
$$\wedge small' = 0$$

If the water doesn't all fit in the big jug, then $5 - big$ gallons are poured out of the little jug.

This definition is fine, but it can be expressed more compactly. Observe that a *SmallToBig* step sets the value of *big* to the smaller of $big + small$ and 5. Let's define *Min* so that $Min(m, n)$ is the smaller of m and n, if m and n are numbers.

$$Min(m, n) \triangleq \text{IF } m < n \text{ THEN } m \text{ ELSE } n$$

```
Min(m,n) == IF m < n THEN m ELSE n
```

Since the amount of water removed from the small jug equals the amount added to the big jug, we can define *SmallToBig* by:

$$SmallToBig \triangleq \wedge big' = Min(big + small, 5)$$
$$\wedge small' = small - (big' - big)$$

This definition has one drawback. When reading an action formula, we often want to see how a particular variable's value changes. This is easiest to do if the value of the primed variable is expressed as a function of the values of the unprimed variables. However, this definition expresses the value of $small'$ in terms of big' as well as of big and $small$. We could fix that by writing the definition as:

$$SmallToBig \triangleq \wedge big' = Min(big + small, 5)$$
$$\wedge small' = small - (Min(big + small, 5) - big)$$

However, it's better not to repeat the expression $Min(big + small, 5)$. I find it more elegant to write the action in terms the amount of water poured from one jug to the other. I prefer writing the action as follows, using the TLA⁺ LET / IN construct□, which allows us to make local definitions within an expression.

$SmallToBig \triangleq$
  LET $poured \triangleq Min(big + small, 5) - big$
  IN $\wedge big' \quad = big + poured$
    $\wedge small' = small - poured$

```
SmallToBig==
  LET poured == Min(big + small, 5) - big
  IN /\big' = big + poured
     /\small' = small - poured
```

(Note that $poured$ equals $Min(small, 5 - big)$.) The definition of the $BigToSmall$ subaction is similar.

$BigToSmall \triangleq$
  LET $poured \triangleq Min(big + small, 3) - small$
  IN $\wedge big' \quad = big - poured$
    $\wedge small' = small + poured$

```
BigToSmall ==
  LET poured == Min(big + small, 3) - small
  IN /\big' = big - poured
     /\small' = small + poured
```

We should also define a type invariant. Clearly, the values of both big and $small$ should be natural numbers, with $big \leq 5$ and $small \leq 3$. To express this, we use the operator $..$ defined in the *Integers* module so that $i .. j$ is the set of integers from i through j. More precisely, $i .. j$ is defined to be the set of all integers k such that $i \leq k$ and $k \leq j$ hold, so $i .. j$ is the empty set if $j < i$. The definition of $..$ is:

$\{x \in S : P(x)\}$ is the subset of S consisting of all its elements x satisfying $P(x)$.□

$$i .. j \triangleq \{k \in Int : (i \leq k) \wedge (k \leq j)\}$$

where Int is defined in the *Integers* module to be the set of all integers. The type invariant is then:

$TypeOK \triangleq \wedge big \quad \in 0 .. 5$
     $\wedge small \in 0 .. 3$

```
TypeOK == /\ big   \in 0..5
          /\ small \in 0..3
```

This definition is best put right after the declaration of the variables big and $small$.

3.2 Applying TLC

Let's now test our spec. Create a new TLC model. Since we used the conventional names for the initial predicate and next-state action, the Toolbox fills in

the What is the behavior spec? section of the model. Add *TypeOK* as an invariant in the What to check? section and run TLC on the model. TLC should find no errors. It will report that the system has 16 distinct reachable states.

The *Die Hard* problem makes learning to write TLA$^+$ specifications a little more fun. But could a TLA$^+$ specification have helped our heroes—especially when they had to solve the problem before a bomb exploded? The answer is yes—at least if they were carrying a computer and were able to write the spec very quickly. They then could have let TLC solve the problem for them.

Remember that their problem was to put 4 gallons of water in a jug, which of course had to be the big jug. All they had to do was have TLC check an invariant asserting that there are not 4 gallons of water in the big jug. Add the invariant *big* $\neq 4$ to your model and run TLC on it. TLC will report that the invariant is violated, and the error trace it produces to demonstrate the violation is a solution to the problem. Moreover, if you select 1 worker thread in the How to run? section of the Model Overview page, TLC will produce a minimal-length error trace. The solution it produces is then one with that takes fewest steps possible—namely, six.

3.3   Expressing the Problem in PlusCal

Although they did solve the problem, the *Die Hard* heroes did not seem to be mathematically sophisticated. They would probably have preferred to write their specification in PlusCal. Let's now see how they could have done that.

Create a new specification called *PDieHard*. The algorithm will use arithmetic operations and the *Min* operator, so copy the EXTENDS statement and the definition of *Min* from the *DieHard* spec and put them at the beginning of module *PDieHard*.

The algorithm is inserted in a comment. It begins with its name, which we take to be *DieHard*, and with a **variables** statement that declares the variables and their initial values. The algorithm looks like this:

The PlusCal keywords **variable** and **variables** are synonyms.

```
--algorithm DieHard {
    variables big = 0, small = 0 ;
    {    body of the algorithm
    }
}
```

We now write the body of the algorithm. The TLA$^+$ specification defines the next-state action *Next* to be the disjunction of six subactions. We first see how to express each of those subactions as a PlusCal statement.

It's easy to express the first four subactions, *FillSmall*, ... , *EmptyBig*. For example, *FillSmall* is expressed by the assignment statement

$$small := 3$$

There's no need to assert that the value of *big* is unchanged. PlusCal is like a very simple programming language in that a statement that does not explicitly change a variable leaves the value of the variable unchanged. (This makes it unlike many real programming languages.)

The *SmallToBig* and *BigToSmall* subactions each have two cases. It's easy to express them with **if** statements. For example, the *SmallToBig* subaction could be described by

$$
\textbf{if } (\ big + small > 5 \ ) \ \{ \ small := small - (5 - big) \,;
$$
$$
big := 5 \qquad\qquad\qquad \}
$$
$$
\textbf{else } \{ \ big := big + small \,;
$$
$$
small := 0 \qquad\qquad \}
$$

As we would expect of a programming language, the order of assignment statements matters. If we changed the order of the two assignments in the **else** clause, the assignment to *big* would be performed with *small* equal to 0, so *big* would be unchanged.

Although this **if** statement correctly describes the *SmallToBig* subaction, it isn't very elegant. It would be nicer to copy the way the subaction is defined in TLA$^+$ and write:

$$
big \quad := big + poured \,;
$$
$$
small := small - poured
$$

where *poured* is defined locally to equal $Min(big+small, 5) - big$. This is written in PlusCal as follows using a **with** statement.

$$
\textbf{with } (\ poured = Min(big + small, \ 5) - big \ )
$$
$$
\{ \ big \quad := big + poured \,;
$$
$$
small := small - poured \quad \}
$$

The *BigToSmall* subaction is described by a similar **with** statement.

In the TLA$^+$ spec, the next-state action is the disjunction of the six subactions, meaning that a step is either a *FillBig* step or a *FillSmall* step or ... or a *BigToSmall* step. Such a disjunction is expressed in PlusCal by an **either / or** statement. So, we can write this disjunction as follows:

$$
\textbf{either } big := 5
$$
$$
\textbf{or} \qquad small := 3
$$
$$
\dots
$$
$$
\textbf{or} \qquad \textbf{with } (\ poured = Min(big + small, \ 3) - small \ )
$$
$$
\{ \ big \quad := big - poured \,;
$$
$$
small := small + poured \ \}
$$

If the body of the algorithm consisted only of this **either / or** statement, an execution of the algorithm would execute the statement once and then halt.

The TLA$^+$ spec describes a system that keeps taking steps forever. To get our algorithm do the same, we put the **either / or** in a **while** (TRUE) loop.

The complete algorithm is here, and the ASCII version is here. Since the PlusCal version lacks the helpful subaction names, I have added comments to explain each clause of the **either / or** statement. (The comments are shaded in the pretty-printed version.)

4 Euclid's Algorithm

Euclid's algorithm is a classic algorithm for computing the greatest common divisor (abbreviated *gcd*) of two positive integers. We consider a simpler and much less efficient version than the one described by Euclid in his *Elements*. However, before writing an algorithm to compute the gcd, we should define precisely what the gcd is.

If you are not familiar with the quantifiers ∀ and ∃, detour here.□

4.1 The Greatest Common Divisor

We want to define an operator *GCD* such that $GCD(m, n)$ equals the gcd of m and n, for numbers m and n. Negative numbers and the number 0 were unknown to Euclid, so let's assume that m and n are positive integers. (The gcd of m and n is undefined if either of them equals 0.) Since we might want to use this operator in some specification other than that of Euclid's algorithm, the instinct of any good engineer is to put the definition into a separate module so it can be re-used. So, let's create a spec to contain the definition of *GCD* and any other related definitions and properties we might need.

In the Toolbox, open a new specification called *GCD*. (TLA⁺ allows the use of the same name for both a module and a defined operator.) You can make it easier to use the module in other specifications by putting it in a separate library folder. Library folders are explained on the help page for the TLA⁺ preferences page.

We'll need the usual operations on integers, so we import them by beginning the module with the statement:

EXTENDS *Integers*   EXTENDS Integers

4.1.1 Divisors

We define the operator *Divides* so that $Divides(p, n)$ equals TRUE if the integer p divides the integer n, and equals FALSE if it doesn't. You learned in grade school that p divides n iff n/p is an integer. The *Integers* module defines *Int* to be the set of all integers. So, an obvious definition of *Divides* is

$$Divides(p, n) \triangleq n/p \in Int$$ Divides(p, n) == n/p \in Int

However, if we use this definition, the Toolbox reports an error because it can't find the definition of the operator $/$.

The *Integers* module is about integers, and n/p is not, in general, an integer. The only arithmetic operations you learned in grade school that the module defines are addition $(+)$, subtraction $(-)$, multiplication $(*)$, and exponentiation

(a^b is typed `a^b`). There is a *Reals* module that defines ordinary division, but it is rarely used because the TLC model checker can't evaluate the operator `/`. So, we define *Divides* using the operators defined in the *Integers* module.

The definition is simple. An integer p divides an integer n iff n equals $q * p$ for some integer q. We can therefore define *Divides* by

$$Divides(p, n) \;\triangleq\; \exists\, q \in Int : n = q * p \qquad\qquad \text{Divides(p, n) == \textbackslash E q \textbackslash in Int : n = q * p}$$

```
Divides(p, n) == \E q \in Int : n = q * p
```

Add this definition and save the module.

Let's test our definition. Create a new TLC model. In it, use TLC to evaluate the expression *Divides*(2, 4). This produces an error message that looks like:

```
TLC encountered a non-enumerable quantifier bound
Int.
line 4, col 27 to line 4, col 29 of module GCD
```

Clicking on the location in the message takes you to the *Int* in the definition.

TLC evaluates an expression of the form $\exists\, x \in S : exp$ by computing all the elements in the set S and evaluating exp for each of those values. It obviously can't do this if S is an infinite set like *Int*.

We don't have to try all integers q to see if there is one satisfying $n = q * p$. Since we're concerned only with positive integers, it's enough to try all integers between 1 and n. So, we could define *Divides* by

$$Divides(p, n) \;\triangleq\; \exists\, q \in 1\,..\,n : n = q * p$$

A principal goal of TLC is that it should not be necessary to modify a spec in order to model-check it. Instead, we let the model tell TLC to override the definition of *Int*, redefining it to equal some finite set of numbers. Have the model redefine *Int* to equal $-1000\,..\,1000$, and run TLC again. This time, TLC's evaluation of *Divides*(2, 4) obtains the value TRUE. Check that TLC calculates *Divides*(2, 5) to equal FALSE.

The gcd of m and n is the largest divisor of both m and n. In other words, it is the maximum of the set of divisors of both m and n. To write this definition mathematically, we first define the set of divisors of a number and the maximum of a set of numbers. The set *DivisorsOf*(n) of divisors of an integer n is obviously:

Recall that $\{x \in S : P(x)\}$ is the subset of S consisting of all its elements x satisfying $P(x)$. $\square$

$$DivisorsOf(n) \;\triangleq\; \{p \in Int : Divides(p, n)\} \qquad\qquad \text{DivisorsOf(n) == \{p \textbackslash in Int : Divides(p, n)\}}$$

```
DivisorsOf(n) == {p \in Int : Divides(p, n)}
```

Add this definition to module *GCD* and have TLC evaluate *DivisorsOf*(493). It should obtain $\{-493, -29, -17, -1, 1, 17, 29, 493\}$.

4.1.2 CHOOSE and the Maximum of a Set

To define the maximum of a set of numbers, we need to introduce the TLA$^+$ CHOOSE operator. The expression

$$\text{CHOOSE } x \in S \ : \ P(x)$$

equals some value v in S such that $P(v)$ equals TRUE, if such a value exists. Its value is unspecified if no such v exists. For example, if we define

$$Foo \ \triangleq \ \text{CHOOSE } i \in Int \ : \ i^2 = 4$$

then *Foo* equals either 2 or -2, since these are the two elements of *Int* whose square equals 4. The semantics of TLA$^+$ do not say which of those two values *Foo* equals. We have absolutely no idea what the value of this expression is:

$$\text{CHOOSE } i \in Int \ : \ i^2 = -4$$

since there is no integer whose square equals -4.

Learn more about CHOOSE here.□

Using CHOOSE, it's easy to define the maximum of a set S of numbers. The maximum of S is an element of S that is greater than or equal to every element of S:

$$SetMax(S) \ \triangleq$$
$$\text{CHOOSE } i \in S \ : \ \forall j \in S \ : \ i \geq j$$

```
SetMax(S)␣==␣
␣␣␣␣CHOOSE␣i␣\in␣S␣:␣\A␣j␣\in␣S␣:␣i␣>=␣j
```

Note that $\geq$ is typed `>=`. It can also be typed `\geq`. Add this definition to module *GCD* and check it by having the Toolbox evaluate the expression *SetMax(DivisorsOf(493))*, which should of course equal 493.

4.1.3 The GCD Operator

The gcd of two positive integers m and n is the maximum of the set of all numbers that are divisors of both of them. That set is just the intersection of their two sets of divisors. We can therefore define:

If you are not familiar with the set operator ∩, detour here□.

$$GCD(m, n) \ \triangleq$$
$$SetMax(DivisorsOf(m) \cap DivisorsOf(n))$$

```
GCD(m,␣n)␣==␣
␣␣␣SetMax(DivisorsOf(m)␣\cap␣DivisorsOf(n))
```

Add this definition to module *GCD* and check that it's correct by evaluating *GCD* for some numbers. You will find that TLC can quickly evaluate the gcd of pairs of numbers less than 1000.

Question 4.1 How can you easily find pairs of numbers whose gcd you know in order to test the definition? ANSWER

This sort of testing will not satisfy a mathematician, but it's good enough for engineers. It checks that we haven't made a gross error, such as misspelling something or writing $\cup$ instead of $\cap$. The only plausible source of error is missing a subtle corner case. We are claiming that this is the correct definition of $GCD(m, n)$ only if m and n are positive integers, so obvious corner cases are (i) if one or both of them equals 1 and (ii) if they are equal. A little thought reveals that there is nothing exceptional about these cases. However, it's a good idea to test them anyway.

4.2 Comments

Mathematics is precise, compact, and elegant. But it's hard to look at a mathematical formula and see what it's about. For example, suppose instead of *Divides*, *DivisorsOf*, *SetMax*, and *GCD*, we had named our operators A, B, C, and D. Their definitions would then look like this.

$$
\begin{aligned}
A(p,\, n) &\triangleq \exists\, q \in Int : n = q * p \\
B(n) &\triangleq \{ p \in Int : A(p,\, n) \} \\
C(S) &\triangleq \text{CHOOSE } i \in S : \forall\, j \in S : i \geq j \\
D(m,\, n) &\triangleq C(B(m) \cap B(n))
\end{aligned}
$$

Imagine how hard it would now be to figure out what these operators mean.

Choosing explanatory names certainly helps, but it's seldom enough to make our specifications easy to understand. We need to add explanatory comments—for example, as in this definition of *Divides*.

$$
Divides(p,\, n) \triangleq \exists\, q \in Int : n = q * p
$$
For integers p and n, equals TRUE iff p divides n.

There are two ways to write comments in TLA$^+$. Text between (* and *) is a comment, and all text that follows a * on the same line is a comment. Thus, the comment above following the definition of *Divides* can be written in either of the following two ways:

```
(* For integers p and n, equals
   TRUE iff p divides n. *)

\* For integers p and n, equals TRUE iff p divides n.
```

Comments can be nested within one another, as in

```
(* This is all (* commented *) text *)
```

Nesting comments is useful for commenting out parts of a specification during testing, but don't do it in actual comments. The pretty-printer ignores comments inside comments. The one exception is that comments inside a PlusCal

algorithm are handled properly, even though the algorithm appears inside a comment.

I like to make comments more visible in the ASCII version by boxing them like this:

```
(*******************************)
(* For integers p and n, equals *)
(* TRUE iff p divides n.         *)
(*******************************)
```

The Toolbox provides commands for writing boxed comments. They are described in the *Editing Comments* section of the *Editing Modules* help page.

The pretty-printer handles boxed comments properly—even if you write something like this.

```
Divides(p, n) ==              (*********************************)
      \E q \in Int :          (* For integers p and n, equals  *)
            n = q * p         (* TRUE iff p divides n -- which  *)
                              (* I think is really neat; don't  *)
                              (* you?                           *)
                              (*********************************)
```

The pretty-printer generally does a reasonably good job of formatting the comments. However, if you want nicely printed comments for others to read, you will have to help it. To find out how, see the Toolbox's *Helping the Pretty-Printer* help page.

Because I explain the specifications in the text as I present them, I will usually omit comments in this hyperbook. You should not omit comments from your specs. Unless you're going to stand next to all the readers of your spec as they read it, and you can project yourself into the future to explain the spec to yourself when you read it a year later, include extensive comments. Every definition and the purpose of every declared variable should be explained in a comment.

Comments are especially important in TLA$^+$ because it is untyped. In a typed language, you would have to declare that the arguments of *Divides* are integers and its value is a Boolean. The absence of type declarations makes the definition shorter and mathematically simpler. However, it imposes on us the responsibility of telling the reader that we expect the arguments to be integers. (It's pretty obvious in this case that the value of $Divides(p, n)$ is a Boolean.)

Text that comes in the file before or after the module is ignored; it can be used to record any information about the spec that you don't want to put in comments within it. The pretty-printer does output this text, but it might not do a very good job of formatting it.

4.3 The Algorithm

Let the positive integers whose gcd we are computing be M and N. Euclid's algorithm uses two variables, which we call x and y. It can be described informally as follows.

- Start with x equal to M and y equal to N.

- Keep subtracting the smaller of x and y from the larger one, until x and y are equal.

- When x and y are equal, they equal the gcd of M and N.

We represent the algorithm in the standard model, describing it in PlusCal.

Open the Toolbox and open a new spec with root module *Euclid*. We'll want to use the definition of GCD, so we want to import it with an EXTENDS statement. Since the GCD module extends the *Integers* module, the EXTENDS statement will also import the *Integers* module. However, I think the spec is easier to understand if it explicitly includes *Integers* in the EXTENDS statement, even if it is redundant. So, we begin the module with

EXTENDS *Integers*, *GCD*   `EXTENDS Integers, GCD`

We need to declare M and N, which we do by writing.

CONSTANTS M, N `CONSTANTS M, N`

The keywords CONSTANT and CONSTANTS are equivalent.

This declares M and N to be unspecified constants—unspecified because we are saying nothing about their values, and constants because their values do not change during the course of a behavior.

We don't want the values of M and N to be totally unspecified; we want them to be positive integers. To assert this assumption, we must express the set of positive integers in TLA$^+$. The *Integers* module defines Nat to be the set of all natural numbers (non-negative integers). The set of positive integers is the set of all natural numbers except 0, which can be written with the set difference operator $\setminus$ as $Nat \setminus \{0\}$. Our assumption about M and N can therefore be written as follows:

ASSUME $\wedge M \in Nat \setminus \{0\}$ `ASSUME␣/\␣M␣\in␣Nat␣\␣{0}`
          $\wedge N \in Nat \setminus \{0\}$ `␣␣␣␣␣␣␣/\␣N␣\in␣Nat␣\␣{0}`

Question 4.2 Use set notation to write this assumption more compactly.   ANSWER

Question 4.3 How many other ways can you write the set of positive integers in TLA$^+$?   ANSWER

As always, the algorithm appears inside a multi-line comment, beginning with the keyword `--algorithm` and followed by the name and an opening `{`. Let's name the algorithm *Euclid*.

```
(***************************************************
--algorithm Euclid {


}
***************************************************)
```

The algorithm uses the two variables x and y, initially equal to M and N, respectively.

variables $x = M$, $y = N$;   ␣variables␣x␣=␣M,␣y␣=␣N␣;

This is followed by the body of the algorithm, enclosed in curly braces.

    Euclid's algorithm works by continually subtracting the smaller of x and y from the larger, stopping when x equals y. If you have used an ordinary programming language, you will probably understand this code, which follows the variable declaration.

```
{ while (x ≠ y) { if (x < y) {y := y − x}
           else        {x := x − y}
      }
}
```

```
␣{␣while␣(x␣#␣y)␣{␣if␣(x␣<␣y)␣{␣y␣:=␣y␣−␣x␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else␣␣␣␣␣␣␣␣{␣x␣:=␣x␣−␣y␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣};
␣}
```

If you don't understand the code, be patient. We'll soon see exactly what it means.

    Having finished the algorithm, you must run the translator to compile it to a TLA$^+$ specification. Do this with the File menu's Translate PlusCal Algorithm command, or by typing control+t. The translator inserts the TLA$^+$ translation after the end of the comment containing the algorithm, between BEGIN TRANSLATION and END TRANSLATION comment lines. If the file already contains such comment lines, the translator replaces everything between those lines with the algorithm's translation.

4.4 The TLA$^+$ Translation

The TLA$^+$ translation describes the precise meaning of the PlusCal algorithm. It begins by declaring the algorithm's variables:

      VARIABLES x, y, pc

The translation has added a new variable pc, which is short for *program control*. The intuitive meaning of the **while** loop is that it continues to execute as long as $x \neq y$ is true. When that formula becomes false, the code following the **while**

loop is executed. In the Standard Model underlying TLA⁺, there is no concept of code. An execution is represented simply as a sequence of states. What code is being executed must be described within the state. In the PlusCal translation, it is described by the value of the variable pc.

After declaring the variables, the translation defines the identifier $vars$ to equal the triple of all the variables.

$$vars \ \triangleq \ \langle x, \ y, \ pc \rangle$$

In TLA⁺, tuples are enclosed between angle brackets $\langle$ and $\rangle$, which are typed << and >>, so the definition of $vars$ is written

```
vars == << x, y, pc >>
```

Next comes the definition of the initial predicate.

$$Init \ \triangleq \ \wedge \, x = M$$
$$\wedge \, y = N$$
$$\wedge \, pc = \text{"Lbl\_1"}$$

The variables x and y have the expected initial values; pc initially equals the string "Lbl_1". We shall see later what this value means and how it was chosen.

The translation next defines Lbl_1 to be the action that describes the steps that can be taken when execution is at the control point "Lbl_1". Such a step represents the execution of a single iteration of the **while** loop. The first conjunct of action Lbl_1 has no primed variables, so it is an enabling condition. It asserts that an Lbl_1 step can occur only when pc equals "Lbl_1", meaning only when control is at the beginning of the **while** statement.

The second conjunct, which is an IF/THEN/ELSE expression, specifies the new values of the three variables x, y, and pc. Let's first look at the new value of pc, which is specified by the value of pc'. If $x \neq y$ is true, then the second conjunct of the outermost THEN clause asserts $pc' = \text{"Lbl\_1"}$. When x and y are not equal, executing one iteration of the **while** statement leaves pc equal to "Lbl_1", meaning that it leaves control at the beginning of the **while**. If $x \neq y$ is false, so x and y are equal, then the first conjunct of the outermost ELSE clause asserts $pc' = \text{"Done"}$, meaning that control is after the **while** loop.

Let's now look at the new values of x and y, which are specified by the values of x' and y'. If $x \neq y$ is true, then these values are specified by the first conjunct of the outermost THEN clause, which is an IF ... THEN ... ELSE expression. This inner IF expression asserts that, if $x < y$ is true, then x' equals x and y' equals $y - x$; otherwise x' equals $x - y$ and y' equals y. If $x \neq y$ is false (so x equals y), then the outermost ELSE clause (of the IF $x \neq y$) asserts UNCHANGED $\langle x, y \rangle$. The built-in TLA⁺ operator UNCHANGED is defined by

$$\text{UNCHANGED } e \ \triangleq \ e' = e$$

for any expression e. Priming an expression e means priming all the variables in e (after fully expanding the definitions of all symbols that occur in e). We therefore have

$$\text{UNCHANGED } \langle x, y \rangle \Leftrightarrow \langle x, y \rangle' = \langle x, y \rangle \quad \text{By definition of UNCHANGED.}$$
$$\Leftrightarrow \langle x', y' \rangle = \langle x, y \rangle \quad \text{By definition of priming an expression.}$$
$$\Leftrightarrow (x' = x) \wedge (y' = y) \quad \text{Because two ordered pairs are equal iff their corresponding elements are equal.}$$

Putting this all together, we see that action Lbl_1 describes a step that

- can occur only when pc equals "Lbl_1".

- if $x \neq y$, subtracts the smaller of x and y from the larger, leaving the smaller of them and pc unchanged.

- if $x = y$, sets pc to "Done", leaving the values of x and y unchanged.

The algorithm begins with pc equal to "Lbl_1". As long as $x \neq y$, it can execute Lbl_1 steps that leave pc equal to "Lbl_1" and decrease x or y. If $x = y$, it can execute an Lbl_1 step that leaves x and y unchanged and sets pc to "Done". When pc equals "Done", the algorithm has terminated and it can do nothing else. We therefore expect Lbl_1 to be the algorithm's next-state action. However, the translation defines $Next$ to be the disjunction of Lbl_1 and another formula. Let's forget about that other formula for now; we'll return to it soon.

The translation then defines two temporal formulas. A temporal formula is a predicate on behaviors (a formula that is true or false of a behavior). Formula $Spec$ is defined to equal $Init \wedge \Box[Next]_{vars}$, where $vars$ is defined to be the triple $\langle x, y, pc \rangle$ of the algorithm's variables. (The formula is written in ASCII as `Init /\ [][Next]_vars`.) We will see later that this temporal formula is true of a behavior iff the behavior is a possible execution of the algorithm. In other words, formula $Spec$ is the TLA$^+$ behavior specification of the algorithm.

The second temporal formula defined by the translation is $Termination$. As we will also see later, it is true of a behavior iff the behavior eventually reaches a state in which pc equals "Done". Hence, formula $Termination$ asserts (of a behavior) that the algorithm terminates.

⎯⎯⎯⎯

You may have remarked that the variable pc did not appear in the translations of our previous PlusCal algorithms: the one-bit clock algorithm $Clock$ and algorithm $DieHard$. The translator is clever enough to realize that control is always at the same point in an execution of those algorithms, so pc is not needed.

4.5 Checking Safety

Correctness of algorithm $Euclid$ means that it satisfies two properties:

- If the algorithm terminates, it does so with x and y both equal to $GCD(M, N)$.

- The algorithm eventually terminates.

The first property is what is called a safety property; the second is a liveness property. We consider the safety property.

The algorithm has terminated iff pc equals "Done". Therefore, the safety property is equivalent to the assertion that the following formula is an invariant of the algorithm (true in all reachable states):

$$(pc = \text{``Done''}) \Rightarrow (x = y) \wedge (x = GCD(M, N))$$

So, let's have TLC check that it is an invariant of the algorithm.

Create a new TLC model for the *Euclid* specification. The Toolbox reports two errors in the model, because the model must specify the values of the declared constants M and N. Double-clicking on a constant in the What is the model? section of the Model Overview page of the model pops up a window in which you can enter the value. (Keep the default Ordinary assignment selection.) Set M to 30 and N to 18.

The Toolbox has set the model's behavior specification to the temporal formula *Spec*. Before checking the invariant, let's just run TLC to make sure there is no error in the algorithm's specification. TLC finds no errors, and reports that there are 6 reachable states and the diameter of the state graph is 5. This is what we expect for an algorithm with a single possible behavior that terminates after taking 5 steps.

Let's now check the invariant. We can enter the invariant directly into the model. However, we might as well put the invariant in a definition in the specification itself. The property of an algorithm that it terminates only with the correct result is called *partial correctness*, so let's add to module *Euclid* the definition:

$PartialCorrectness \triangleq$
$\quad (pc = \text{``Done''}) \Rightarrow (x = y) \wedge (x = GCD(M, N))$

```
PartialCorrectness␣==
␣␣(pc␣=␣"Done")␣=>␣(x␣=␣y)␣/\␣(x␣=␣GCD(M,␣N))
```

Add the invariant *PartialCorrectness* to the Invariants part of the What to check? section of the Model Overview page and run TLC. This produces an error, with the not very helpful error message

```
Evaluating invariant PartialCorrectness failed.
```

The error trace shows that this error occurred when TLC was evaluating the invariant on the last state of a complete execution. This is the first state TLC computed in which $pc = $ "Done" equals true, so it is the first state in which it had to compute $GCD(M, N)$ when evaluating *PartialCorrectness*. TLC can't evaluate $GCD(M, N)$ unless we override the definition of *Int* to make it a finite set. As we did for the *GCD* spec, use the Definition Override section of the Advanced Options page to have the model redefine *Int* to equal $-1000 .. 1000$.

TLC should now find no error, verifying that the algorithm terminated with x and y equal to $GCD(M, N)$.

Try changing the values of M and N and running TLC again. Each run should take a couple of seconds for values of M and N less than 1000. Since we know that Euclid's algorithm is correct, checking a few values of M and N will give us confidence that our PlusCal version is correct.

If we didn't know that Euclid's algorithm was correct, we would need to check it for many more values. Instead of checking that our algorithm computes the gcd of M and N, let's check that it computes the gcd of all pairs of numbers in $1 .. N$. We do this by declaring the initial values of x and y to be arbitrary elements of $1 .. N$. We also add two variables $x0$ and $y0$ that initially equal x and y, respectively, and whose values are left unchanged. We then check that, when the algorithm terminates, x and y equal $GCD(x0, y0)$.

This is one situation where there is no good way to test the algorithm without modifying it.

Change the **variables** declaration of the algorithm to:

variables $x \in 1 .. N$, $y \in 1 .. N$, $x0 = x$, $y0 = y$;

Rerun the translator and examine the formulas *Init* and *Next* that it produces. Formula *Init* should be what you expect it to be, and formula *Next* is the same as before except for a conjunct asserting that $x0$ and $y0$ are unchanged.

Create a new model by cloning the model you already created. In the model's Invariants section, uncheck the invariant *PartialCorrectness* and add the invariant:

$$(pc = \text{"Done"}) \Rightarrow (x = y) \wedge (x = GCD(x0, y0))$$

When you're not sure how long checking a model will take, start with a very small model. Set the value of N to be 5, so there are 25 possible behaviors of the algorithm (because there are 25 different initial states). Even with such a small model, running TLC with a single worker thread takes 30 seconds on my computer. Why is it so slow?

TLC is spending almost all its time computing $GCD(x0, y0)$ when evaluating the invariant. Doing that requires it to compute $Divisors(x0)$ and $Divisors(y0)$. TLC computes $Divisors(n)$ from the definition of $Divisors$ by enumerating all the elements p in *Int* and checking if $Divides(p, n)$ is true. In the common case when p does not divide n, this computing $Divides(p, n)$ requires TLC to check that n does not equal $p * q$ for every element q of *Int*. Since our model redefines *Int* to be a set with about 2000 elements, computing $GCD(x0, y0)$ requires TLC to compute an expression of the form $n = p * q$ about 8 million times. It computes $GCD(x0, y0)$ 25 times for this model—once for the final state of each of the behaviors. Experimentation reveals that there is a constant 7 second start-up overhead, and simple arithmetic then shows that it takes TLC a little over .1 microsecond to compute $n = p * q$. This is about 100 times longer than it takes a Java program to evaluate the same expression on my computer.

All the positive divisors of a positive integer n are elements of $1 \ldots n$. TLC will therefore correctly compute $GCD(x0, y0)$ for $x0$ and $y0$ in $1 \ldots N$ if we redefine *Int* to equal $1 \ldots N$. Change the model to override the definition of *Int* with this value. It now takes TLC only 7 seconds to run the model on my computer for $N = 5$. For $N = 100$, it takes 33 seconds.

This example illustrates that for checking a spec, it helps to have a basic understanding of how TLC works. It also shows that the simplicity and elegance of mathematics compared to programming languages comes at a high price in efficiency of execution. Fortunately, checking all behaviors of a small model is generally more effective at finding errors in an algorithm than checking randomly chosen behaviors of a programming-language implementation.

Instead of checking the algorithm by adding an invariant to the model, we can add an **assert** statement to the algorithm. Place the following statement right after the **while** statement:

assert $(x = y) \wedge (x = GCD(x0, y0))$ `assert (x = y) /\ (x = GCD(x0, y0))`

Execution of the statement **assert** P does nothing if P is true, and it reports an error if P is false. Save the module and run the translator again. If you followed the directions above exactly, this will yield a translator error reporting a missing semicolon (;) before the **assert**. Separate PlusCal statements must be separated by semicolons. (A semicolon can be placed at the end of a sequence of statements, but it is not required.) Insert the missing semicolon, which most people place just to the right of the **}** that ends the **while** statement. Save the module and run the translator again. This should result in the parser error:

      `Unknown operator: 'Assert'.`

The translation of the **assert** statement uses a special operator *Assert* defined in the standard *TLC* module. It defines $Assert(P, m)$ to equal TRUE if P equals TRUE. If TLC evaluates P to be different from TRUE, it reports an error that includes m. (In that case the value of P shouldn't matter.) Add *TLC* to the EXTENDS statement and save the module. The parser error disappear, and you can now run TLC.

Try changing the **assert** statement to cause an error—for example change $x = y$ to $x \neq y$—and run TLC. Clicking on the appropriate links in the error message will take you to the **assert** statement and to its translation.

4.6   Checking Liveness

Open the model for the *Euclid* algorithm that you have been checking with TLC. Open the Properties part of the What to check? section of the Model Overview page. It should list the property *Termination*, but with it unchecked. Remember that *Termination* is the temporal formula that is true of a behavior

iff the behavior terminates (reaches a state with *pc* equal to "Done"). (If it's not in the list, add it.) Check that property to tell TLC to check it. Have the model set N to 10 and run TLC on it.

TLC reports that

```
Temporal properties were violated.
```

and it produces an error trace consisting of a single state, which is a possible initial state (one satisfying the *Init* predicate), followed by the mysterious indication ⟨Stuttering⟩. This trace describes a behavior consisting of a single state, representing an execution that stops in an initial state. (It will become clear later why the trace says *Stuttering*.)

A behavior of the algorithm is a sequence $s_1 \to s_2 \to \cdots$ that satisfies two conditions:

1. *Init* is true if the variables have their values in state s_1. (Remember that a state is an assignment of values to variables.)

2. For any pair $s_i \to s_{i+1}$ of successive states, *Next* is true if the unprimed variables have their values in s_i and primed variables have their values in s_{i+1}.

It seems natural also to require that the behavior doesn't end before it has to—in other words, to add the condition:

3. The behavior does not end in a state s_n if there exists a state s_{n+1} such that the sequence $s_1 \to \ldots \to s_{n+1}$ also satisfies condition 2.

However, the PlusCal algorithms we have written thus far do not have this requirement. They allow all behaviors that satisfy conditions 1 and 2, including behaviors that stop in the initial state. More precisely, the temporal formulas *Spec* that are those algorithms' translations allow all such behaviors.

To add requirement 3 for the behaviors of an algorithm, instead of beginning the algorithm with `--algorithm`, we begin it with:

```
--fair algorithm
```

Make this change, run the translator, and run TLC again on the model. This time, TLC finds no error, verifying that for the model, all behaviors terminate.

Examining the translation, we find that the new definition of the behavior specification *Spec* is the conjunction of its original definition and the formula $\mathrm{WF}_{vars}(Next)$ (written in ASCII as `WF_vars(Next)`). It is this formula that expresses condition 3. The requirement is called *weak fairness* of the action *Next*. We will study fairness formulas later. For now, you need only know that this particular formula, with *Next* the specification's next-state action, asserts condition 3.

Why don't we require condition 3 to hold for all algorithms?

4.7 The Translation Revisited

Let's return to the definition of *Next* in the translation, which is

$$Next \triangleq Lbl\_1 \lor (pc = \text{"Done"} \land \text{UNCHANGED } vars)$$

where *vars* is defined to equal $\langle x, y, x0, y0, pc \rangle$. Action *Lbl_1* describes the steps allowed by the body of the algorithm. The second disjunct allows steps that start in a state in which *pc* equals "Done" and leaves the algorithm's five variables unchanged. A step that leaves all of a specification's variables unchanged is called a *stuttering* step.

The comment added by the translator tells us that this disjunct is added to prevent deadlock on termination. To verify that it's needed, comment out the disjunct, save the module, and run TLC on the same model. (An easy way to comment out those two lines is to select them and type control+/.) Indeed, TLC reports that deadlock was reached and shows an error trace ending in a terminated state.

TLC considers a reachable state from which there is no next state satisfying the next-state action to be a deadlock error. The only difference between deadlock and termination is that termination is deadlock that we want to happen—or equivalently, that deadlock is termination we don't want to happen. TLC doesn't know whether or not we wanted this deadlock to happen. We can tell TLC to ignore deadlock by unchecking the *Deadlock* box in the *What to check* section of the model overview page. However, it's possible to write Plus-Cal algorithms that can deadlock at a state with $pc \neq$ "Done". This usually indicates an error—that is, deadlock that we didn't want to happen—so we want TLC to report it. Therefore, the translation adds this disjunct to the next-state action so TLC doesn't treat termination as deadlock.

4.8 The Grain of Atomicity

The TLA$^+$ translation defines the next-state action *Next* for which an execution of one iteration of the **while** loop is a single step. Why? Why didn't the translator produce a definition of *Next* in which evaluating the **while** test and executing the body of the **while** statement are represented as two separate steps? Perhaps it should have made execution of the **if** statement two steps, one evaluating the condition and the second executing either the **if** or the **else** clause.

In PlusCal, what constitutes a step is specified by the use of labels in the code. A step is execution from one label to the next. For uniprocessor algorithms like the ones we have written so far, we can omit the labels and let the translator decide where they belong. For algorithm *Euclid*, the translator decided that there should be a single label *Lbl_1* on the **while** statement. To see that this is the case, let's explicitly add the label *abc* to the **while** loop, so it becomes:

abc: **while** ($x \neq y$) { ...

Run the translator. The translation is exactly the same as before except that formula Lbl_1 has become formula abc, whose definition is the same as the original definition of Lbl_1 except that the string "Lbl_1" has been replaced by "abc".

There are rules for where labels must go and where they may not go. Most of the rules serve to make the translation simple, which is important because we want to be able to reason about it. You'll learn the rules as we go along, and the translator's error messages will tell you if you've omitted a necessary label or put one where it shouldn't go. The first two rules are:

- The first statement in the body of the algorithm must have a label.

- A **while** statement must have a label.

Both imply that the translator had to add a (virtual) label where it did. If we let it decide where the labels should be, it uses as few as possible. This produces a specification in which an execution has the fewest possible steps, which makes model checking most efficient. It also produces the simplest translation. For uniprocess algorithms, we usually care only about the answer they produce and not what constitutes a step.

Let's see what happens when we add another label. Put the label d on the **if** statement, so the body of the algorithm becomes:

$$abc: \textbf{while } (\ x \neq y \ ) \ \{ \ d: \textbf{if } (\ x < y \ ) \ \{ \ y := y - x \ \} $$
$$\textbf{else} \qquad \{ \ x := x - y \ \} $$
$$\} \ ; $$
$$\textbf{assert } (x = y) \wedge (x = GCD(x0, \ y0)) $$

There are two kinds of steps in an execution of this algorithm:

An abc step: The step starts with control at abc and, based on the value of the test $x \neq y$, either moves control to d or else executes the **assert** statement and moves control to $Done$ (the implicit control point at the end of the algorithm).

A d step: A step that starts with control at d, executes the **if** step, and then moves control to abc.

Run the translator. The translation defines two subactions, abc and d, that describe these two kinds of steps. It defines $Next$ to be the disjunction of these two subactions and of the subaction allowing stuttering steps when the algorithm has terminated.

Try adding other labels in addition to or instead of d. Make sure you understand the translations. In this algorithm, you can add a label at the beginning of any complete statement. The only requirement is that the **while** statement

be labeled. As you have already figured out, the translation defines a subaction for each label. Run TLC on the different versions (for a small value of N) and compare their numbers of reachable states.

4.9 Why Euclid's Algorithm Is Correct

Checking an algorithm with TLC can give us some confidence that an algorithm is correct. How much confidence depends on the algorithm. It cannot show us *why* the algorithm is correct. For that, we need a proof.

In this track, we write only informal correctness proofs. Writing any kind of proof helps you understand an algorithm and therefore helps you avoid errors. However, it's often easy to write an incorrect informal proof that claims to prove a property that an algorithm doesn't satisfy—especially for a safety property. The informal safety proofs we will write can be made as rigorous as necessary to give us sufficient confidence in their correctness. (What constitutes sufficient confidence depends on what the algorithm is going to be used for.) If necessary, they can be turned into formal TLA$^+$ proofs and checked with the TLAPS proof system. Few readers will ever need to write a formal proof. However, learning to write formal proofs will improve your ability to write rigorous informal ones. I therefore urge you to learn how to write and check formal proofs by reading at least the beginning of the TLA$^+$ Proof Track$^\square$.

Since we are reasoning about the algorithm, not testing it, let's use the simpler, original version of the algorithm. Recall that this version computed the gcd of M and N with x and y the only (declared) variables, and it had no labels and no **assert** statement. Change the algorithm in the *Euclid* module back to that version and run the translator.

4.9.1 Proving Invariance

The safety property we want to prove about algorithm *Euclid* is the invariance of the state predicate *PartialCorrectness*, which is defined to equal

$$(pc = \text{"Done"}) \implies (x = y) \wedge (x = GCD(M, N))$$

A state predicate is a formula that is true or false of a state. In other words, it is a Boolean-valued expression that may contain variables but no primes (or temporal operators). Invariance of a state predicate means that it is true in every state of every behavior of the algorithm. To prove that a state predicate *Inv* is true in every state of a particular behavior $s_1 \to s_2 \to \ldots$, we prove:

1. *Inv* is true in state s_1.

2. For every step $s_n \to s_{n+1}$ in the behavior, if *Inv* is true in state s_n then it is true in state s_{n+1}.

It follows by induction from 1 and 2 that *Inv* is true for every state s_n of the behavior. This reasoning shows that we can prove that *Inv* is true in every state of every behavior by proving:

1. *Inv* is true for any initial state, and

2. If *Inv* is true in a state s and $s \rightarrow t$ is a possible step of the algorithm, then *Inv* is true in state t.

An initial state is one that satisfies the initial predicate *Init*. Therefore the first condition is equivalent to the truth of:

I1. *Init* $\Rightarrow$ *Inv*

A step $s \rightarrow t$ is a possible step of the algorithm only if the next-state action *Next* is true when each unprimed variable has its value in state s and each primed variable has its value in state t. For any state predicate P, we define P' to be the formula obtained from P by priming all the variables in it. For example, *PartialCorrectness'* equals

$$(pc' = \text{``Done''}) \Rightarrow (x' = y') \wedge (x' = GCD(M, N))$$

Condition 2 is then satisfied if the following formula is true:

I2. *Inv* $\wedge$ *Next* $\Rightarrow$ *Inv'*

Make sure you understand why the truth of I2 implies the truth of condition 2 above.

An invariant *Inv* satisfying I1 and I2 is called an *inductive invariant* of the algorithm. (A predicate satisfying I2 is sometimes called an inductive invariant of the next-state action *Next*.) Although *PartialCorrectness* is an invariant of algorithm *Euclid*, it is not an inductive invariant. It satisfies I1 but not I2. For example, consider the following values for the unprimed and primed variables:

$$x = 42 \quad y = 42 \quad pc = \text{``Lbl\_1''} \quad x' = 42 \quad y' = 42 \quad pc' = \text{``Done''}$$

You can check that *Next* is true for these values of the primed and unprimed variables by substituting them in the definition of *Lbl_1* and checking that the resulting formula equals TRUE. This is perhaps easier to see by observing that the step

$$
\begin{bmatrix} x & = & 42 \\ y & = & 42 \\ pc & = & \text{``Lbl\_1''} \end{bmatrix}
\rightarrow
\begin{bmatrix} x & = & 42 \\ y & = & 42 \\ pc & = & \text{``Done''} \end{bmatrix}
$$

which starts with control at the beginning of the **while** statement and ends with control at the end of the algorithm, is allowed by the code in the algorithm's body. With those values of the primed and unprimed variables,

PartialCorrectness equals TRUE (because $pc = Done$ equals FALSE), and *PartialCorrectness'* equals the formula $42 = GCD(M, N)$ (because $pc' =$ "Done" and $x' = y'$ both equal TRUE). Hence with these substitutions, I2 becomes TRUE $\land$ TRUE $\Rightarrow (42 = GCD(M, N))$, which equals $42 = GCD(M, N)$. Thus, I2 is false for *Inv* equal to *PartialCorrectness* unless the gcd of M and N happens to equal 42. In that case, we can replace 42 by another number to get an example in which I2 is false. Therefore, *PartialCorrectness* is not an inductive invariant.

This was a long calculation to demonstration something that should have been obvious. Formula *PartialCorrectness* is true in any state with pc not equal to "Done". Its truth tells us nothing about the relation between the values of x, y, and $GCD(M, N)$ during the algorithm's execution, so its truth during the execution can't imply that it will be true upon termination. However, doing this long calculation should help you understand that, by describing the algorithm with two formulas, *Init* and *Next*, TLA$^+$ reduces reasoning about an algorithm to simple mathematics.

We are still left with the problem of proving the invariance of *PartialCorrectness*. We do that by finding an inductive invariant *Inv* that, in addition to I1 and I2, satisfies:

I3. *Inv* $\Rightarrow$ *PartialCorrectness*

Conditions I1 and I2 imply that *Inv* is true in all reachable states, which by I3 implies that *PartialCorrectness* is true in all reachable states, so it is an invariant.

The fundamental reason why Euclid's algorithm computes the gcd is that it maintains the invariance of the state predicate:

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

This is an inductive invariant of the algorithm. However, it doesn't satisfy I3 because it doesn't imply that x equals y on termination. An inductive invariant *Inv* that satisfies $I3$ is:

$$Inv \triangleq \land\ GCD(x, y) = GCD(M, N)$$
$$\land\ (pc = \text{"Done"}) \Rightarrow (x = y)$$

The proof that *Inv* satisfies I1–I3 requires three facts about the gcd. These facts, which we call $GCD1$–$GCD3$, are expressed by the following theorems:

THEOREM $GCD1 \triangleq \forall m \in Nat \setminus \{0\} : GCD(m, m) = m$

THEOREM $GCD2 \triangleq \forall m, n \in Nat \setminus \{0\} : GCD(m, n) = GCD(n, m)$

THEOREM $GCD3 \triangleq \forall m, n \in Nat \setminus \{0\} : (n > m) \Rightarrow (GCD(m, n) = GCD(m, n - m))$

Let's just assume them for now; we'll return to them later.

for a proof of the invariance of *PartialCorrectness*. The first thing you will notice is that this proof doesn't look like an ordinary mathematician's proof. Instead, it is hierarchically structured. Proofs of algorithms can be quite complicated, and the way to handle complexity is by hierarchical structure. Here are some other things to observe about the proof style.

- A proof is either a leaf proof, consisting of a short paragraph; or it is a sequence of steps, each with a proof, ending with a QED step.

- A QED step asserts the goal of the current level of proof. Its proof shows that this goal is proved by the preceding steps.

- A CASE statement asserts that the current proof's goal is true if the CASE assumption is.

Learning to write proofs that are correct and easy to read is an art. Here are a couple of tips.

- If a leaf proof is too long to be easily understood, it should be decomposed into a non-leaf proof, adding another level to the hierarchy. A leaf proof that is not easy to understand could easily be incorrect.

- Any previous proof steps required by a leaf proof should be explicitly mentioned, as should other significant facts being used (such as *GCD1*–*GCD3*).

This proof may seem rigorous. Actually it is incorrect—for a reason that should eventually become obvious to you. Throughout the proof, there is an implicit assumption that x, y, M, and N are positive integers. The ASSUME statement in the module asserts that M and N are positive integers, justifying that assumption. Step 1 is therefore correct, though its proof should mention that it uses the assumption. However, there is nothing in the hypotheses of any other step to imply that x and y are positive integers—an assumption that is needed to apply *GCD1*–*GCD3*. We can't even prove that x and y are numbers.

In all the proofs except for that of step 1, we get to assume that *Inv* is true. Thus, we can justify the assumption that x and y are positive integers by having *Inv* assert it. Let's do that by defining:

$$TypeOK \;\triangleq\; \begin{aligned}&\wedge\; x \in Nat \setminus \{0\}\\ &\wedge\; y \in Nat \setminus \{0\}\end{aligned}$$

and changing the definition of *Inv* to

$$Inv \;\triangleq\; \begin{aligned}&\wedge\; TypeOK\\ &\wedge\; GCD(x,y) = GCD(M,N)\\ &\wedge\; (pc = \text{``Done''}) \Rightarrow (x = y)\end{aligned}$$

47

With this change, our proof becomes correct in the sense that the assertion made by every step is true. However, a more rigorous proof would mention that the proof uses *TypeOK*. Also, in steps 2.1–2.3, the proofs of *Inv$'$* need to prove *TypeOK$'$*.

In general, an inductive invariant must contain a type-correctness condition. Since that's not a very interesting part of the invariant, we encapsulate this condition in a separate formula that I like to call *TypeOK*. The formula usually has a conjunct for each variable, asserting that the variable is an element of some set. For uniprocess PlusCal algorithms such as this one, there may be no need of a type-correctness condition for the variable *pc*. We may not bother mentioning the use of *TypeOK* in an informal proof. However, we should include it in the inductive invariant, because proving statements that are not true is a bad habit to get into.

Question 4.4 Modify the algorithm by labeling the **while** loop *abc* and labeling   ANSWER
the **if** statement *d*. Show that the formula *Inv* defined above is not an inductive invariant of the resulting algorithm. Find an inductive invariant of this algorithm that implies *PartialCorrectness*.

Click here if you already learned how to prove partial correctness of programs.

4.9.2 Verifying *GCD1*–*GCD3*

A complete proof of Euclid's algorithm should include a proof of *GCD1*–*GCD3*. However, before we do any proving, we should use TLC to check the correctness of these theorems. It's a lot easier to prove something if it's true. And even an obviously true theorem could be incorrect because of a typo.

Open the *GCD* spec in the Toolbox and create a new model. We can check all three theorems at once by having the model tell TLC to evaluate the constant expression

$$\langle\, GCD1,\ GCD2,\ GCD3 \,\rangle \hspace{4cm} \texttt{<<GCD1, GCD2, GCD3>>}$$

We saw in Section 4.1 that we must override the definition of *Int* with a finite set of integers to allow TLC to evaluate the *GCD* operator. The three theorems are all of the form

$$\forall \ldots\, \in Nat \setminus \{0\} \,:\, \ldots$$

TLC can evaluate such formulas only if we override the definition of *Nat*. Have the model override the definitions of *Nat* and *Int* with small sets of integers—for example, $0 \ldots 5$—and run TLC on it. If you've made no error, it should report the value $\langle\, TRUE,\ TRUE,\ TRUE \,\rangle$. You can then check *GCD1*–*GCD3* on a larger model. It should take TLC one or two minutes to do this for a model that defines *Nat* and *Int* to equal $0 \ldots 100$.

Having checked $GCD1$–$GCD3$ with TLC, we can now think about proving them. Theorems $GCD1$ and $GCD2$ follow easily from the definition of GCD and we won't bother proving them. The proof of $GCD3$ uses this simple fact

> **Lemma Div** For any integers m, n, and d, if d divides both m and n then it also divides both $m + n$ and $n - m$.

You should have no trouble proving it.

Here is a proof of $GCD3$. The structure of the proof becomes clearer, and the proof easier to read, if we introduce notation to replace some of the prose, obtaining this proof. Compare the two proofs. To help you understand the second proof, here it is with comments. Although the notation may seem strange, you should be able to see that it makes the second proof easier to read. Ease of reading is very important for complex proofs.

The formal TLA$^+$ versions of the invariance proof of Euclid's algorithm and the proof of $GCD1$—$GCD3$ are in Section 11 of the Proof Track$^\square$.

4.9.3 Proving Termination

To prove that algorithm *Euclid* always terminates (assuming fairness), we observe that each step of the algorithm that doesn't reach a terminating step decreases either x or y and leaves the other unchanged. Thus, such a step decreases $x + y$. Since x and y are always positive integers, $x + y$ can be decreased only a finite number of times. Hence, the algorithm can take only a finite number of steps without terminating.

In general, to prove that an algorithm terminates, we find an integer-valued state function W for which:

- $W \geq 0$ in any reachable, non-terminating state.

- If s is any reachable state and $s \to t$ is any step satisfying the next-state action, then either the value of W in state s is greater than its value in state t, or the algorithm is terminated in state t.

To reason about reachable states, we use an invariant—which by definition is a predicate that is true in every reachable state of an algorithm. Let *Next* be the algorithm's next-state action. The same kind of reasoning that led to condition I2 above shows that we can prove termination by finding a state function W and an invariant I of the algorithm satisfying:

L1. $I \Rightarrow (W \in Nat) \vee (pc = \text{"Done"})$

L2. $I \wedge Next \Rightarrow (W > W') \vee (pc' = \text{"Done"})$

The state function W is called a *variant function*. For algorithm *Euclid*, we let W be $x + y$ and we let I be the (inductive) invariant *Inv*.

The use of the ordering $>$ on natural numbers in this method can be generalized to any well-founded ordering on a set. However, the generalization is seldom needed to prove termination of uniprocess algorithms.

4.10   Euclid's Algorithm for Sets

We now consider a generalization of Euclid's algorithm that I find elegant. It computes the gcd of a set of numbers, rather than of just two numbers. We start by defining $SetGCD(T)$ to be the gcd of a set T of positive integers. It equals the maximum of the set of numbers that divide all the numbers in T:

$$SetGCD(T) \triangleq SetMax(\{d \in Int : \forall t \in T : Divides(d, t)\})$$

Add the definition to module *GCD* and check it for one or two small sets of positive integers. (Use the same model you did before, which defined *Int* to be a finite set of integers.)

The algorithm, which computes the gcd of a non-empty set *Input* of positive integers, uses a single variable S. Its informal description is:

- Start with S equal to *Input*.

- While S has more than one element, choose elements x and y in S with $y > x$, remove y from S and insert $y - x$ in S.

- If S contains a single element, that element is $SetGCD(Input)$.

(If you don't understand how removing one element from S and inserting another could reduce the number of elements in S, you need to read about sets☐.)

The body of the PlusCal algorithm should be a **while** loop whose test asserts that S has more than one element. The standard *FiniteSets* module defines $Cardinality(S)$ to equal the number of elements in S, if S is a finite set. The value of $Cardinality(S)$ is unspecified if S is not a finite set. We will assume that *Input* is a finite set, so S will always be a finite set. The **while** loop's test can therefore be written as $Cardinality(S) > 1$. Here is the complete body of the algorithm:

```
while ( Cardinality(S) > 1 )
   { with (  x ∈ S,  y ∈ {s ∈ S : s > x}  )
       {  S := (S \ {y}) ∪ {y − x}  }
   }
```

To understand the meaning of the **with** statement, let's look at the translation of the algorithm's body:

$$Lbl\_1 \triangleq \land pc = \text{``Lbl\_1''}$$
$$\land \text{IF } Cardinality(S) > 1$$
$$\text{THEN } \land \exists\, x \in S :$$
$$\exists\, y \in \{s \in S : s > x\} :$$
$$S' = ((S \setminus \{y\}) \cup \{y - x\})$$
$$\land pc' = \text{``Lbl\_1''}$$
$$\text{ELSE } \land pc' = \text{``Done''}$$
$$\land S' = S$$

The colored formula is the translation of the **with** statement; the green formula is the translation of its body, the assignment to S. Let's examine the formula piece by piece.

$S \setminus \{y\}$

> The set obtained by removing all the elements in the set $\{y\}$ from S—in other words, obtained by removing y from S.

$(S \setminus \{y\}) \cup \{y - x\}$

> The set obtained from S by removing y and inserting $y - x$.

The green formula therefore asserts that S' (the new value of S) equals the set obtained by removing y from (the old value of) S and inserting $y - x$.

$\{s \in S : s > x\}$

> The set of elements in S that are greater than x.

The entire formula therefore asserts that there exist x and y in S, with $y > x$, such that the green formula is true. The meaning of the **with** statement is therefore:

> Execute the body with x an arbitrary element of S and y an arbitrary element of S greater than x.

In general, the statement

$$\textbf{with } (\ v_1 \in S_1,\ \ldots\ ,\ v_1 \in S_1\ )\ \{\ \Sigma\ \}$$

with ($v \in S$) versus
with ($v = S$)

is executed by letting each v_i be arbitrary element in S_i and executing Σ with those values of the v_i. TLC will check the executions obtained by all possible choices of the v_i.

Create a new module named *SetEuclid* that EXTENDS module *GCD* and *Integers*. Enter the PlusCal specification, translate it, and test that TLC executes the specification on a model with *Init* a small set of positive integers. Make it a fair algorithm (beginning with `--fair algorithm`) and have TLC check that it terminates.

Partial correctness of Euclid's algorithm, which asserts that on termination S contains the single element *SetGCD(Input)*, is expressed by the invariance of:

$$PartialCorrectness \triangleq (pc = \text{``Done''}) \Rightarrow (S = \{SetGCD(Input)\})$$

Add this definition to module *SetEuclid* and have TLC check that the invariance of *PartialCorrectness*. (Since evaluating this formula requires TLC to compute *SetGCD(Input)*, your model will have to override the definition of *Int*.)

The proof of partial correctness is analogous to that of algorithm *Euclid* and is based on the invariance of

$$SetGCD(S) = SetGCD(Input)$$

A rigorous proof uses the inductive invariant

$$
\begin{aligned}
SInv \triangleq\ & \wedge\ TypeOK \\
& \wedge\ SetGCD(S) = SetGCD(Input) \\
& \wedge\ PartialCorrectness
\end{aligned}
$$

where the type invariant is defined by

$$
\begin{aligned}
TypeOK \triangleq\ & \wedge\ S \subseteq Nat \setminus \{0\} \\
& \wedge\ S \neq \{\} \\
& \wedge\ IsFiniteSet(S)
\end{aligned}
$$

The assumption that S is finite is required because we don't know what the expression $Cardinality(S)$ in the **while** test means if S is not a finite set. To prove that *TypeOK* is true in the initial state, we need the assumption:

$$
\begin{aligned}
\text{ASSUME} \triangleq\ & \wedge\ Input \subseteq Nat \setminus \{0\} \\
& \wedge\ Input \neq \{\} \\
& \wedge\ IsFiniteSet(Input)
\end{aligned}
$$

Question 4.5 Rewrite the algorithm without using the *Cardinality* operator, so   ANSWER
partial correctness is true even for infinite sets *Input*. (The rewritten algorithm
obviously does not terminate if *Input* is an infinite set.)

The proof of termination is based on the observation that each non-terminating step of the algorithm decreases the sum of all the elements of S. To state this rigorously, we must define the sum of the elements in a finite set of numbers. The only way I know to define this mathematically is with a recursive definition (also called by mathematicians an inductive definition):

- The sum of the empty set is 0.

- The sum of a non-empty set T of integers is the sum of some element t in T plus the sum of the elements in $T \setminus \{t\}$.

Since it doesn't matter what element t of T is chosen in the recursive step, we can use the CHOOSE operator to select it. The obvious definition is then:

$$SetSum(T) \;\triangleq\; \text{IF } T = \{\} \text{ THEN } 0$$
$$\text{ELSE } \text{LET } t \;\triangleq\; \text{CHOOSE } x \in T : \text{TRUE}$$
$$\text{IN} \quad t + SetSum(T \setminus \{t\})$$

Add the definition to module *SetEuclid* and save the module. This produces a parsing error complaining that the operator *SetSum* is undefined when used on the right-hand side of the symbol. In TLA$^+$, a symbol must be defined or declared before it can be used. To allow such a recursive definition of *SetSum*, the definition must be preceded by this RECURSIVE declaration:

RECURSIVE *SetSum*(_)   `RECURSIVE SetSum(_)`

The declaration should be put right before the definition. The parser should now accept the specification. You can use TLC to check that this is a correct definition of the sum of a finite set of integers.

To prove termination, we prove conditions L1 and L2 with the invariant I equal to *SInv* and the variant function W equal to *SetSum*(S). The informal proof is straightforward. It uses the following fact about *SetSum*:

$$\forall\, T \in \text{SUBSET } Nat :$$
$$IsFiniteSet(T) \Rightarrow \wedge \forall\, t \in T : SetSum(T \setminus \{t\}) = SetSum(T) - t$$
$$\wedge \forall\, t \in Nat : SetSum(T \cup \{t\}) \leq SetSum(T) + t$$

where SUBSET *Nat* is the set of all subsets of the set *Nat* of natural numbers. A rigorous proof reveals that some simple facts about finite sets and *Cardinality* are also required.

Question 4.6   Show that the correctness of algorithm *SetEuclid* implies the following important result from number theory. The gcd of a set $\{n_1, \ldots, n_k\}$ of positive integers equals $i_1 * n_1 + \ldots + i_k * n_k$ for some integers i_j.   ANSWER

If you want to learn how to write formal TLA$^+$ proofs, you can now start reading the Proof track.$^\square$

5   The Generalized Die Hard Problem

We now generalize the *Die Hard* problem of Section 3 to the problem of obtaining an arbitrary quantity of water with an arbitrary collection of jugs. The main purpose of this example is to introduce the use of functions.

5.1   The PlusCal Representation

Open a new spec named *DieHarder* and add an EXTENDS statement to import the *Integers* module. As in the original *DieHard* spec, we'll need the operator *Min*, where $Min(m, n)$ is the smaller of the two numbers m and n. So, add its definition to the module.

$Min(m, n) \triangleq$ IF $m < n$ THEN m ELSE n `Min(m,n) == IF m < n THEN m ELSE n`

A specification of the general Die Hard problem requires two constant parameters:

> *Jugs*   The set of jugs.
>
> *Capacity*   A value that describes the capacity of each jug.

For later use, we also declare the constant *Goal*, which will represent the number of gallons of water that our generalized heros must obtain. So, add the following declaration to the *DieHarder* module:

CONSTANTS *Goal*, *Jugs*, *Capacity*   `CONSTANTS Goal, Jugs, Capacity`

Splitting a module into sections.

We let *Capacity* describe the capacities of the jugs by making it a function. Mathematical functions appear in programming languages, where they are called *arrays*. What programming languages call the *index set* of an array is known to mathematicians as the *domain* of a function. However, while programming languages limit what kind of set can be the index set of an array, mathematics allows a function to have any set as its domain—including an infinite set. Programmers think of an array A as a collection of "containers", one for each element in its index set; they think of $A[i]$ as the contents of A's container i. Mathematicians think of a function A as a rule that assigns to each element i in its domain a value $A(i)$. TLA$^+$ uses the notation of programmers in writing the value of a function A applied to an element i as $A[i]$ rather than as $A(i)$. In all other ways, I will use the language of mathematicians, writing about functions rather than arrays.

In the C language, the index set of an array can only be a set of the form $0 .. n$ for some integer n. Too many language designers have copied this limitation of C.

   The constant *Capacity* will be a function whose domain is the set *Jugs* of all jugs. For each jug j, the value of $Capacity[j]$ should be a positive integer—that is, an element of $Nat \setminus \{0\}$. We say that *Capacity* is a *function from Jugs*

to $Nat \setminus \{0\}$. The set of all such function is written $[Jugs \rightarrow Nat \setminus \{0\}]$. We obviously want *Goal* to be a natural number. We therefore add the following "type assumption" to the module:

ASSUME $\wedge\ Goal \in Nat$
$\qquad\quad \wedge\ Capacity \in [Jugs \rightarrow Nat \setminus \{0\}]$

```
ASSUME␣/\␣Goal␣\in␣Nat
␣␣␣␣␣␣␣␣/\␣Capacity␣\in␣[Jugs␣->␣Nat␣\␣{0}]
```

We will specify the system of jugs and water in PlusCal. The system's state is described by the contents of the jugs. We represent it with a variable *injug*, where $injug[j]$ is the number of gallons of water in jug j. The declaration of variable *injug* must specify its initial value. Initially, $injug[j]$ equals 0 for each jug j. In other words, initially *injug* equals a function with domain *Jugs* that assigns 0 to every element in its domain. Mathematics provides no standard notation for writing this function. It is written in TLA$^+$ as $[j \in Jugs \mapsto 0]$. In general, the expression

$$[v \in S \mapsto e(v)]$$

equals a function f with domain S such that $f[v] = e(v)$ for every v in S. Thus, the algorithm begins

> **--algorithm** *DieHarder* **{**
> **variable** $injug = [j \in Jugs \mapsto 0]$ **;**

Had we wanted all the jugs to be full in the initial state, we would have declared the initial value of *injug* to be $[j \in Jugs \mapsto Capacity[j]]$.

From our original *DieHard* algorithm, we see that the body of algorithm *DieHarder* should be a **while** (TRUE) loop whose body is an **either**/**or** statement that permits three different actions:

- Filling some jug j.

- Emptying some jug j.

- Pouring from some jug j to another jug k.

It should be clear how to express the first two actions as PlusCal statements:

with ($j \in Jugs$) `fill jug j`
  { $injug[j] := Capacity[j]$ **}**

with ($j \in Jugs$) `empty jug j`
  { $injug[j] := 0$ **}**

Remembering the **with** statement in the body of algorithm *SetEuclid* and how we represented pouring from one jug to another in algorithm *DieHard*, we see that pouring from a jug j to a jug k is represented by this PlusCal statement.

with ($j \in Jugs, k \in Jugs \setminus \{j\}$) `pour from jug j to jug k`
      { with (*poured* =

Warning: this code has an error.

$$Min(injug[j] + injug[k], \, Capacity[k]) - injug[k] \,)$$
```
                    {  injug[j] := injug[j] − poured ;
                       injug[k] := injug[k] + poured
                    }
               }
```

Add the ASCII code of the algorithm to a comment in the module, save the module, and run the translator. The translator reports the following error:

> ```
> Second assignment to same variable inside a 'with' statement at
> line ...
> ```

Clicking on the error message takes you to the second assignment statement inside the code representing pouring from jug j to jug k. The message is telling us that we can't have two assignment statements within the body of a **with** statement that assign to the same variable. To understand why not, you need to know two additional rules for labels in a PlusCal algorithm.

- Two separate assignment statements that assign to the same variable cannot occur within a single step.

- The body of a **with** statement cannot contain a label.

Recall that a step consists of execution from one label to the next. The first rule implies that a label must come between any two assignment statements to the same variable. In this case, that means that the assignment to $injug[k]$ must have a label. However, the second rule says that a label cannot go there. Hence, these two rules imply that it is impossible for the translator to assign labels to this algorithm.

The solution to this problem is to replace the semi-colon (;) between the assignment statements with || (two | characters), turning the two separate assignment statements into this single multi-assignment:

$$injug[j] \, := \, injug[j] - poured \, \parallel injug[k] \, := \, injug[k] + poured$$

A multi-assignment statement consists of a sequence of assignments of the form:

$$v_1 \, := \, e_1 \, \parallel \, \ldots \, \parallel \, v_n \, := \, e_n$$

It is executed by evaluating all the expressions e_i, and then simultaneously performing the assignments to all the v_i. For example, if v and w are variables, then executing

$$v \, := \, w \, \parallel \, w \, := \, v$$

interchanges the values of the two variables, setting the new value of v to the old value of w and vice-versa. A multi-assignment statement is most often used to assign to multiple "components" of a function in the same step, as in the *DieHarder* algorithm.

After turning the two assignment statements into one multi-assignment, the module should look like this. The translator should now produce no more errors.

5.2   Checking the Algorithm

We now use TLC to check algorithm *DieHarder*. Let's create a model for which algorithm *DieHarder* is equivalent to algorithm *DieHard*—that is, one with two jugs of capacities 3 and 5, and a goal of 4. Open a new model and have it set *Goal* to 4.

We want the model to assign some set of two elements to *Jugs*. We could let those elements be two numbers such as 3 and 47, or two strings such as "bigJug" and "smallJug". However, when some part of a spec can have any value, it's usually a good idea to have the model assign it a *model value*. A model value is treated by TLC to be a value about which it knows nothing, except that it is unequal to any other model value. We can give a model value any legal identifier name. Since we're modeling the original Die Hard system, let's make the elements of *Jugs* be the model values *big* and *small*. On the dialog for assigning a value to the constant *Jugs*, type {*big*, *small*} into the text field and select the Set of model values option. Click on Next and then on Finish.

To learn more about model values, see the *Model Values and Symmetry* Toolbox help page.

We want the model to assign to *Capacity* a function with domain {*big*, *small*} such that *Capacity*[*big*] = 5 and *Capacity*[*small*] = 3. One way of doing this is to assign it the value

$$[j \in Jugs \mapsto \text{IF } j = big \text{ THEN } 5 \text{ ELSE } 3]$$

However, when creating a model, we can write this function more conveniently as follows:

$$(big :> 5) @@ (small :> 3) \qquad \text{(big :> 5) @@ (small :> 3)}$$

(Select the Ordinary assignment option when assigning the value to *Capacity*. The assignment of the set of model values {*big*, *small*} to *Jugs* declares *big* and *small* to be model values in this model.) It should be clear how to use this notation to write any function with a finite domain. The operators :> and @@, which are defined in the standard *TLC* module, are explained in Section 15.2$^\square$. (Operators defined in the *TLC* module can be used in creating a model, even if the spec doesn't import that module.)

Add the formula

$$\forall j \in Jugs : injug[j] \neq Goal$$

to the model's list of invariants to be checked. Running TLC should produce as an error trace a behavior that solves the Die Hard problem for this selection of jugs. Try other models. Start with a model with two jugs of capacity 3 and 6 gallons, having the goal of obtaining 4 gallons of water. TLC will report that the alleged invariant actually is an invariant. For that model, the problem has no solution.

Question 5.1   Under what condition does the generalized Die Hard problem   ANSWER have a solution? That is, for what values of the constants is the formula above not an invariant of algorithm *DieHarder*?

5.3   The TLA$^+$ Translation

Let's now consider the TLA$^+$ translation of the algorithm. Let's start with the **either** clause, which describes the action of filling a jug.

> **with** ($j \in Jugs$)   `fill jug j`
>   **{** $injug[j] := Capacity[j]$ **}**

We know that its translation is $\exists j \in Jugs : \Sigma$, where Σ is the translation of the assignment statement. What is the translation of the assignment statement? Most people think it should be $injug[j]' = Capacity[j]$. Let's see if it is.

In the Toolbox, run TLC on a model that produces an error trace showing a solution to the Die Hard problem. The first step in that trace is one that fills a jug. Double-click on the row <Action line ... between the two states of that first step, which takes you to the part of the next-state action satisfied by that step. Holding down the control control key while double-clicking takes you to the corresponding place in the PlusCal code, which is indeed the **either** clause. As you can see, the translation of the assignment statement is not $injug[j]' = Capacity[j]$. Why not?

Recall the body of algorithm *Euclid* and its TLA$^+$ translation. Observe that the translation of the assignment statement $y := y - x$ is not the formula $y' = y - x$. That formula describes the new value of y, but says nothing about the new value of x. The translation of the assignment statement is the formula

$$\land\ y' = y - x$$
$$\land\ x' = x$$

that also describes the new value of x.

Similarly, the translation of the statement $injug[j] := Capacity[j]$ must describe not only the new value of $injug[j]$, but the values of $injug[k]$ for all k in the domain of $injug$. The formula $injug[j]' = Capacity[j]$ says nothing about the new value of any expression other than $injug[j]$. Not only does it say nothing about the value of $injug[k]'$ for $k \neq j$, it doesn't even say anything about the domain of $injug'$. There are lots of values of $injug'$ that satisfy this formula—for example, it is satisfied if $injug'$ is the function

$$[k \in \{j\} \mapsto Capacity[j]]$$

whose domain contains only the single element j.

The translation of $injug[j] := Capacity[j]$ must be a formula asserting that $injug'$ is a function that is exactly the same as the function $injug$, except that $injug'[j]$ equals $Capacity[j]$. We can write that function as:

$$[k \in Jugs \mapsto \text{IF}\ k = j\ \text{THEN}\ Capacity[j]\ \text{ELSE}\ injug[k]]$$

The domain of an arbitrary function f can be written in TLA$^+$ as DOMAIN f. We can therefore also write the function above as

$$[k \in \text{DOMAIN}\ injug \mapsto \text{IF}\ k = j\ \text{THEN}\ Capacity[j]\ \text{ELSE}\ injug[k]]$$

Since assignments to arrays appear frequently in specifications, TLA$^+$ provides a more convenient way of writing this function. As you can see from the translation of algorithm *DieHarder*, it can be written as

$$[injug \text{ EXCEPT } ![j] = Capacity[j]]$$

A further examination of the algorithm's translation shows that the multi-assignment statement

$$injug[j] := injug[j] - poured \ \| \ injug[k] := injug[k] + poured$$

is translated as

$$injug' = [\, injug \text{ EXCEPT } ![j] = injug[j] - poured,$$
$$![k] = injug[k] + poured\,]$$

In general, the expression $[f \text{ EXCEPT } ![x] = d, \ ![y] = e]$ is defined to equal $[[f \text{ EXCEPT } ![x] = d] \text{ EXCEPT } ![y] = e]$. The meaning of the further generalization $[f \text{ EXCEPT } ![x_1] = e_1, \ \ldots, \ ![x_n] = e_n]$ should be clear.

Question 5.2 Explain why changing the body of the **while** loop of algorithm *DieHarder* to the following produces an equivalent algorithm.   ANSWER

```
with ( j ∈ Jugs )
  { either { injug[j] := Capacity[j] }
    or      { injug[j] := 0 }
    or      with ( k ∈ Jugs \ {j} )
              { with ( poured =
                          Min(injug[j] + injug[k], Capacity[k]) − injug[k] )
                { injug[j] := injug[j] − poured ||
                  injug[k] := injug[k] + poured
                }
              }
  }
```

6 Alternation

6.1 The Problem

We now begin the subject that concerns most of the *Principles* track of this hyperbook: multiprocess algorithms and systems. We start with *alternation*, which is the simplest form of multiprocess synchronization. In alternation synchronization, two processes each have an operation to perform, and they must execute those operations alternately.

Let's call the processes the *producer* and the *consumer*, and let their operations be called *put* and *get*, respectively. The two processes must cooperate to perform the sequence of operations:

$$put \;\rightarrow\; get \;\rightarrow\; put \;\rightarrow\; get \;\rightarrow \ldots$$

Think of the *put* operation as putting an object into a box, and the *get* operation as taking the object out of the box and doing something with it.

To express the problem in our Standard Model, we let the variable *box* represent the state manipulated by the operations. For simplicity, we represent the *put* and *get* operations by these two PlusCal statements

$$box \;:=\; Put(box) \qquad\qquad box \;:=\; Get(box)$$

Question 6.1 This representation makes the *put* and *get* operations deterministic. Given the initial value b_0 of *box* in a behavior, the sequence of values of *box* in the behavior must be   ANSWER

$$[box = b_0] \;\rightarrow\; [box = Put(b_0)] \;\rightarrow\; [box = Get(Put(b_0))] \;\rightarrow\; [box = Put(Get(Put(b_0)))] \;\rightarrow\; \ldots$$

How can we modify the representation to allow the *put* and *get* operations to be nondeterministic?

We can declare *Put* and *Get* to be parameters of a specification with the statement

CONSTANTS $Put(\_)$, $Get(\_)$

However, each TLC model that we use would then have to assign particular operators to *Put* and *Get*. For convenience, we define specific operators *Put* and *Get*.

I could probably think of hundreds of sensible ways to define *Put* and *Get*. For reasons that may (or may not) become clearer in a later section, I like to define them so the value of *box* is a tuple that initially equals the 0-tuple $\langle\,\rangle$. We define *Put* to be the operator that appends the value "widget" to a tuple, so

$$Put(\langle e_1, \ldots, e_n \rangle) = \langle e_1, \ldots, e_n, \text{"widget"} \rangle$$

We define *Get* to be the operator the removes the first element of a tuple, so

$$Get(\langle e_1, \ldots, e_n \rangle) = \langle e_2, \ldots, e_n \rangle$$

Thus, a behavior should have the following sequence of values of *box*:

$$[box = \langle \, \rangle] \; \rightarrow \; [box = \langle \text{"widget"} \rangle] \; \rightarrow \; [box = \langle \, \rangle] \; \rightarrow \; [box = \langle \text{"widget"} \rangle] \; \rightarrow \; \ldots$$

These definitions make it easy to have TLC check that an algorithm does execute the two operations alternately. Executing two successive *put* operations sets *box* to a tuple containing two or more elements. If that doesn't happen, then executing two successive *get* operations causes the second one to try to remove the first element of a 0-tuple, which will produce a TLC error. Having TLC check the invariant that the tuple *box* has at most one element will therefore produce an error if the two operations are not executed alternately.

In TLA$^+$, tuples are the same as finite sequences, an n-tuple being a sequence of length n. We define *Put* and *Get* using the operators *Append* and *Tail* defined in the standard *Sequences* module$^\square$. The definitions are:

$$Put(s) \;\triangleq\; Append(s, \text{"widget"}) \qquad\qquad \texttt{Put(s) == Append(s, "widget")}$$
$$Get(s) \;\triangleq\; Tail(s) \qquad\qquad\qquad\qquad\quad \texttt{Get(s) == Tail(s)}$$

The *Sequences* module also defines the operator *Len* so that $Len(s)$ is the length of a sequence s. We can have TLC check that an algorithm implements alternation by checking the invariance of $Len(box) \le 1$.

6.2 The One-Bit Clock Revisited

The simplest way to implement alternation is with a one-bit clock that starts at 0. The *put* operation is performed when the clock changes from 0 to 1, and the *get* operation is performed when it changes from 1 to 0. So, let's return to our first example: the one-bit clock. We again represent the value of the clock with a variable b. However, this time we represent the clock as a two-process algorithm: a *Tick* process that changes the value of b from 0 to 1, and a *Tock* process that changes it from 1 to 0. The algorithm begins as usual, with the declaration of variable b setting its initial value to 0.

--**algorithm** *TickTock* {
    variable $b = 0$;

In PlusCal, a process has both a name and an *identifier* (*id* for short), which can be any value. We let the process named *Tick* have id 0 and the one named *Tock* have id 1. Here is the code for the two processes. The order of the two **process** declarations makes no difference. By default, the translator requires us to provide labels for a multiprocess algorithm. Since a step consists of execution from one label to the next, the single label in each process makes an entire execution of its **while** loop's body a single step.

Intuitively, the statement **await** $b = 0$ causes process *Tick* to wait until b equals 0 before it can execute the rest of the step. However, that intuitive explanation shouldn't be taken too seriously. The real meaning of an **await** statement is revealed by its TLA$^+$ translation.

Open a new specification named *TickTock* and insert into it the ASCII version of the algorithm. Run the translator, and then create a model and run TLC on it. There should be no error. TLC reports as expected that there are two reachable states. Now examine the initial predicate *Init* and the next-state action *Next* generated by the TLA$^+$ translation. The predicate *Init* obviously is defined to equal $b = 0$ (with a superfluous $\wedge$). The action *Next* is the disjunction of the two subactions *Tick* and *Tock*. From their definitions, we see that *Next* equals

$$
\begin{aligned}
&\vee \ \wedge \ b = 0 \\
&\quad\ \wedge \ b' = 1 \\
&\vee \ \wedge \ b = 1 \\
&\quad\ \wedge \ b' = 0
\end{aligned}
$$

This is the action *Next1* that we defined above in our first specification of the one-bit clock.

The Toolbox's Goto PCal Source command allows you to find the PlusCal code corresponding to parts of the TLA$^+$ translation. Select a region of the translation and run the command either from the module editor's right-click menu or by typing F10. The command jumps to and highlights the PlusCal source. If you try that on the translation, you will discover that the statement **await** P simply adds the conjunct P to the appropriate part of the next-state action.

Question 6.2 Write a uniprocess PlusCal algorithm whose translation defines ANSWER
exactly the same definitions of *Init* and *Next* as that of algorithm *TickTock*.

Is the one-bit clock a uniprocess system or a multiprocess system? It's neither. The number of processes is a property of the syntactic representation of a system in PlusCal or in some programming language. It is not a property of the system itself. The true meaning of (a model of) a system lies in the mathematics: the initial predicate and next-state action (plus perhaps a fairness formula). As we saw in our original discussion of the one-bit clock, there are many ways to write the initial predicate and next-state action to produce the same behavior specification—that is, to produce equivalent temporal formulas $Init \wedge \Box[Next]_{vars}$. As shown by Question 6.2, different PlusCal code can even produce syntactically identical behavior specifications.

6.3 Specifying Alternation: Safety

We now specify alternation synchronization by adding to the one-bit clock specification the *put* and *get* operations, expressed using the *Put* and *Get* operators defined above. The specification is in module *Alternation*.

The module first imports the standard *Integers* and *Sequences* modules and defines *Put* and *Get*. Then comes the behavioral specification, written as a PlusCal algorithm called *Alternate*. The algorithm declares the variable b of the one-bit clock, with initial value 0, and the variable *box* used by the *put* and *get* operations, with initial value equal to the empty sequence.

The producer and consumer processes are specified by **process** declarations, which are the same as the the two clock processes except with the added assignments to *box* that represent the *put* and *get* operations. The labels have also been changed. Create the *Alternation* specification in the Toolbox, using this ASCII version of the module's body.

It should be clear that the only behaviors of this algorithm are the infinite behavior

$$
\begin{bmatrix} b & = & 0 \\ box & = & \langle\,\rangle \end{bmatrix} \rightarrow \begin{bmatrix} b & = & 1 \\ box & = & \langle \text{``widget''} \rangle \end{bmatrix} \rightarrow \begin{bmatrix} b & = & 0 \\ box & = & \langle\,\rangle \end{bmatrix} \rightarrow \begin{bmatrix} b & = & 1 \\ box & = & \langle \text{``widget''} \rangle \end{bmatrix} \rightarrow \cdots
$$

and any finite prefix of it. (We haven't specified any fairness requirement yet, so execution can stop at any point.) Hence, except for fairness, the sequence of values of *box* specifies the proper alternation of the *put* and *get* operations. Let's use TLC to check that this is the case, and that we haven't made a mistake. Create a new model, add to it the invariant $Len(box) \leq 1$, and run TLC on it. TLC reports that the invariant is satisfied, and that there are just two reachable states, as expected.

There are two problems with this specification. I've mentioned the first: that we haven't specified any fairness requirements. The spec therefore describes only the safety properties of alternation, not any liveness properties. Liveness is discussed below.

The second problem is that I informally described alternation in terms of *put* and *get* operations, which are represented in terms of the single variable *box*. However, algorithm *Alternation* also specifies the values of the variable b. The variable b serves only to "control" the possible values of the variable *box*. We don't care about its value. It's possible to write a philosophically correct specification that hides the variable b, but we won't bother with philosophy. Instead, we take a practical approach that depends on what it means to implement the specification. This approach is discussed later.

How to hide variables in TLA.

6.4 Specifying Alternation: Liveness

We saw in Section 4.6 that the initial predicate *Init* and next-state action *Next* assert the following safety properties of a behavior $s_1 \rightarrow s_2 \rightarrow \cdots$.

1. *Init* is true in s_1. (Remember that a state is an assignment of values to variables.)

2. Each step $s_i \rightarrow s_{i+1}$ of the behavior is a *Next* step.

Recall that condition 2 means that formula *Next* is true when each unprimed variables is replaced by its values in state s_i and each primed variable is replaced by the variable's value in state s_{i+1}.

Beginning the algorithm with `--fair algorithm` adds the fairness requirement $\mathrm{WF}_{var}(Next)$, which asserts:

3. The behavior does not end in a state s_n if there exists a state s_{n+1} such that the sequence $s_1 \rightarrow \ldots \rightarrow s_{n+1}$ also satisfies condition 2.

We say that an action A is *enabled* in a state s iff there exists a state t such that $s \rightarrow t$ is an A step. Assuming the truth of condition 2, we can restate condition 3 as:

3a. The behavior does not end in a state s_n in which *Next* is enabled.

For algorithm *Alternate*, the next-state action *Next* is enabled in every reachable state, so weak fairness of *Next* implies that it has no finite behaviors. It must execute forever.

This same liveness condition can be expressed as weak fairness of each of the processes. For an arbitrary action A, the weak fairness requirement $\mathrm{WF}_{var}(A)$ is defined to assert the following two conditions for the behavior $s_1 \rightarrow s_2 \rightarrow \cdots$.

3a. The behavior does not end in a state s_n in which A is enabled.

3b. If the behavior is infinite, then there is no n such that the infinite behavior $s_n \rightarrow s_{n+1} \rightarrow \cdots$ has no A steps but A is enabled in all of its states.

If A is the next-state action *Next*, then condition 3b is implied by condition 2, which asserts that every step of the behavior is a *Next* step.

Preceding the keyword **process** by **fair** causes the translator to conjoin weak fairness of that process's next-state action to the specification *Spec*. For example, change the *Producer* process's declaration in algorithm *Alternate* so it begins:

> **fair process** (*Producer* $= 0$)

and run the translator. This produces the definition

$$Spec \triangleq \begin{aligned} &\wedge Init \wedge \square[Next]_{vars} \\ &\wedge \mathrm{WF}_{vars}(Producer) \end{aligned}$$

The action *Producer* is the next-state action of process *Producer*, which describes an execution of the body of the process's **while** loop. That action is enabled in a state s iff s assigns the value 0 to b. Conditions 1 and 2 imply that every second step of a behavior is a *Producer* step. Hence, with this definition, *Spec* asserts that a behavior cannot stop in a state with $b = 0$. Similarly, fairness of the *Consumer* process implies that a behavior cannot stop in a state with $b = 1$. Fairness of both processes therefore implies that a behavior must be

infinite, so it is equivalent to fairness of the next-state action. The equivalence of weak fairness of all processes and weak fairness of the next-state action for a system specification indicates a special class of system in which stopping one process causes the entire system to stop.

What liveness requirement do we want for alternation synchronization? There is no single answer. In some applications, the producer and consumer represent two components of a system that we want to model as running forever. In that case, we want fairness of both processes—or equivalently, fairness of the entire algorithm's next-state action. In other applications, the producer represents a client whose *put* operations are requests, and the consumer represents a server whose *get* operations are responses. In that case, we usually require the server to respond to each request, but don't require the client to keep sending requests. The requirement that the server responds to each request is expressed by fairness of the *Consumer* process, since that implies that a *Consumer* step must occur after every *Producer* step.

Let's ignore fairness for now. Eliminate all fairness from the algorithm and run the translator.

6.5   The Two-Phase Handshake Protocol

We now consider a common hardware signaling protocol for implementing alternation synchronization called the *two-phase handshake.* As illustrated here,
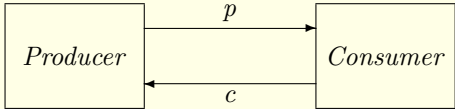
the processes communicate using two one-bit "wires" p and c, where p is set by the producer and read by the consumer, and c is set by the consumer and read by the producer.

We describe the algorithm in terms of the operator $\oplus$, which is defined by: $\oplus$ is typed `(+)` or `\oplus`.

$$a \oplus b \;\;\triangleq\;\; (a + b) \,\%\, 2$$

We are interested in applying $\oplus$ only to elements in the set $\{0, 1\}$, for which its table of values is shown in the margin. When restricted to the set $\{0, 1\}$, the operator $\oplus$ is called *exclusive or.* Observe that if a equals 0 or 1, then $a \oplus 1$ is the "complement" of a.

$0 \oplus 0 = 0 \quad 0 \oplus 1 = 1$
$1 \oplus 0 = 1 \quad 1 \oplus 1 = 0$

We represent the two-phase handshake protocol as algorithm *Handshake* in the *Handshake* module. Compare the bodies of the processes' **while** loops in algorithm *Handshake* with those in algorithm *Alternation*:

	Handshake	Alternation
Producer:	**await** $p = c$; $box := Put(box)$; $p := p \oplus 1$	**await** $b = 0$; $box := Put(box)$; $b := 1$
Consumer:	**await** $p \neq c$; $box := Get(box)$; $c := c \oplus 1$	**await** $b = 1$; $box := Get(box)$; $b := 0$

Question 6.3 Write down the single infinite behavior of algorithm *Handshake*. ANSWER
In each of the states of that behavior, write down the value of $p \oplus c$.

Create specification *Handshake* in the Toolbox, using this ASCII version of its
body, and run the PlusCal translator on it. Have TLC check that the formula
$Len(box) \leq 1$ is an invariant of the algorithm, as it should be for an implemen-
tation of alternation synchronization with these definitions of *Put* and *Get*.

6.6 Refinement

Compare the answer to Question 6.3, with the behavior of algorithm *Alternation*.
You will see that the values of $p \oplus c$ in the behavior of algorithm *Handshake*
equal the values of b in the corresponding behavior of algorithm *Alternation*.
We can view algorithm *Handshake* as implementing the variable b of algorithm
Alternation with the expression $p \oplus c$. It also implements the variable *box* of
algorithm *Alternation* with the expression *box*. (Algorithm *Handshake* is thus
implementing the variable *box* of the *Alternation* algorithm with its own variable
box.)

Let $\overline{b}$ and $\overline{box}$ be the expressions (containing the variables of *Handshake*)
that implement the variables of algorithm *Alternation*:

$$\overline{b} \triangleq p \oplus c \qquad \overline{box} \triangleq box$$

Let's write a behavior of algorithm *Handshake* showing the values of $\overline{b}$ and $\overline{box}$
in all the states.

$$
\begin{bmatrix} p &=& 0 \\ c &=& 0 \\ box &=& \langle\,\rangle \\ \overline{b} &=& 0 \\ \overline{box} &=& \langle\,\rangle \end{bmatrix}
\rightarrow
\begin{bmatrix} p &=& 1 \\ c &=& 0 \\ box &=& \langle\,\text{"widget"}\,\rangle \\ \overline{b} &=& 1 \\ \overline{box} &=& \langle\,\text{"widget"}\,\rangle \end{bmatrix}
\rightarrow
\begin{bmatrix} p &=& 1 \\ c &=& 1 \\ box &=& \langle\,\rangle \\ \overline{b} &=& 0 \\ \overline{box} &=& \langle\,\rangle \end{bmatrix}
\rightarrow
$$

$$
\begin{bmatrix} p &=& 0 \\ c &=& 1 \\ box &=& \langle\,\text{"widget"}\,\rangle \\ \overline{b} &=& 1 \\ \overline{box} &=& \langle\,\text{"widget"}\,\rangle \end{bmatrix}
\rightarrow
\begin{bmatrix} p &=& 0 \\ c &=& 0 \\ box &=& \langle\,\rangle \\ \overline{b} &=& 0 \\ \overline{box} &=& \langle\,\rangle \end{bmatrix}
\rightarrow \cdots
$$

If we delete the values of the variables p, c, and box and erase the overbars from $\overline{b}$ and $\overline{box}$, we get the sequence

$$
\begin{bmatrix} b & = & 0 \\ box & = & \langle\,\rangle \end{bmatrix} \rightarrow
\begin{bmatrix} b & = & 1 \\ box & = & \langle\,\text{``widget''}\,\rangle \end{bmatrix} \rightarrow
\begin{bmatrix} b & = & 0 \\ box & = & \langle\,\rangle \end{bmatrix} \rightarrow
\begin{bmatrix} b & = & 1 \\ box & = & \langle\,\text{``widget''}\,\rangle \end{bmatrix} \rightarrow
$$

$$
\begin{bmatrix} b & = & 0 \\ box & = & \langle\,\rangle \end{bmatrix} \rightarrow \cdots
$$

which is a behavior of algorithm *Alternation*. The definitions of $\overline{b}$ and $\overline{box}$ are called a *refinement mapping* from *Handshake* to *Alternation*, and we say that algorithm *Handshake* implements algorithm *Alternation* under this refinement mapping.

I now generalize what we've done for the particular algorithms *Handshake* and *Alternation* to two arbitrary specifications H and A. To do this, I need to distinguish between the two meanings of the term *specification*. In the following discussion, a *specification* S is a TLA$^+$ module, together with any modules that it imports. This module must define a *behavior specification*, which is a temporal formula (usually named *Spec*). A state of S is an assignment of values to the variables declared in S; it is not necessarily a reachable state of S's behavior specification. A behavior of S is any sequence of states of S; it does not necessarily satisfy S's behavior specification.

A *refinement mapping* from a specification H to a specification A is an assignment of an expression $\overline{v}$ to each variable v of A, where $\overline{v}$ is defined in terms of the variables of H. This refinement mapping defines, for each state s of H, the state $\overline{s}$ of A that assigns to each variable v of A the value of $\overline{v}$ in state s. If σ is the behavior $s_1 \rightarrow s_2 \rightarrow \cdots$ of H, we define the behavior $\overline{\sigma}$ of A to be $\overline{s_1} \rightarrow \overline{s_2} \rightarrow \cdots$. We say that H *implements* A under this refinement mapping iff, for each behavior σ satisfying the behavior specification of H, the behavior $\overline{\sigma}$ satisfies the behavior specification of A.

Having precisely defined the meaning of implementation under a refinement mapping, I will resume the informal use of the term *specification*. If you are confused by any informal statement that I write, translate it into a more rigorous one that clearly distinguishes between modules and their behavior specifications.

We have seen that *Handshake* implements *Alternation* under the refinement mapping defined above. Since the refinement mapping defines $\overline{box}$ to equal box, this implies that for any behavior σ allowed by algorithm *Handshake*, the behavior $\overline{\sigma}$ allowed by algorithm *Alternation* has the same sequence of values of box. Since correctness of an alternation synchronization depends only of the values assumed by box, this implies that *Handshake* implements alternation synchronization. In general, we define an algorithm to implement alternation synchronization (for these particular definitions of *Put* and *Get*) to mean that it implements algorithm *Alternation* under a refinement mapping that defines $\overline{box}$ to equal box.

As the following problem shows, whether or not it is interesting that one specification implements another under a refinement mapping depends very much on the refinement mapping. In general, we can define correctness as implementation under a refinement mapping satisfying some condition. However, we are often content to show that a system or algorithm satisfies certain desired properties.

Question 6.4 Let *Count* be a specification with a single variable n whose behavior specification allows the single infinite behavior

$$[n = 1] \; \rightarrow \; [n = 2] \; \rightarrow \; [n = 3] \; \rightarrow \; [n = 4] \; \rightarrow \; \cdots$$

Show that if A is any specification whose behavior specification allows an infinite behavior, then *Count* implements A under some refinement mapping.

Question 6.5 Show how to express the property that a formula I is an invariant of a specification S as the property that S implements a specification under a refinement mapping.

We can use TLC to check if one specification implements another under a refinement mapping. In module *Handshake*, after the algorithm's translation, add this statement which we say *instantiates* module *Alternation*.

$$A \;\; \triangleq \;\; \text{INSTANCE } \textit{Alternation} \text{ WITH } b \leftarrow p \oplus c, \; box \leftarrow box$$

We'll see later what this statement means. For now, just observe that the WITH part describes the refinement mapping. If a variable v of the instantiated module does not appear in a WITH clause, then the clause $v \leftarrow v$ is assumed. Thus, the clause $box \leftarrow box$ can be eliminated from this statement. We could eliminate the entire WITH part by preceding the statement with the definition

$$b \;\; \triangleq \;\; p \oplus c$$

We can use any identifier in place of A (as long as it's not already defined or declared).

To have TLC check that algorithm *Handshake* implements algorithm *Alternation* under this refinment mapping, open a model and, in the Properties subsection of the What to check? section of the Model Overview page, add the property $A!Spec$. This tells TLC to check that the *Handshake* specification with its behavior specification *Spec* (indicated by the model), implements the *Alternation* specification with its behavior specification *Spec*, under the refinment mapping described by the INSTANCE statement. Run the model. TLC should find no error, confirming that *Handshake* implements *Alternation* under the refinement mapping.

Now, modify the algorithm of module *Alternation* by making the *Consumer* process a **fair process**. Run the translator on that module and run TLC again on the same model (of specification *Handshake*). This time, TLC should report:

```
Temporal properties were violated.
```

and should produce an error trace that halts after the second state. (Again, stopping is indicated by the mysterious <Stuttering>, which will soon be explained.) This behavior corresponds under the refinement mapping to a behavior that stops after the producer takes a step, which is not allowed by the fairness requirement for process *Consumer* in algorithm *Alternation*.

Add fairness to process *Consumer* of algorithm *Handshake* and run the translator. TLC should now confirm that *Handshake* (with fairness of the consumer) implements *Alternation* (with fairness of the consumer) under the refinement mapping. Fairness of the *Consumer* process of algorithm *Handshake* rules out the behavior of the algorithm found by TLC showing that *Handshake* (without consumer fairness) did not implement *Alternation* (with consumer fairness) under the refinment mapping.

Detour Deriving the handshake protocol from the alternation specification.

6.7   Refinement and Stuttering

6.7.1   Adding Steps

In a more accurate model of the two-phase handshake protocol, execution of one iteration of a process's **while** loop would consist of multiple separate steps. Let's split each iteration into two steps: the first executing the **await** statement, the second executing the two assignment statements. We do this by adding a label to each process as follows:

$$
\begin{array}{ll}
p1: \textbf{while (TRUE)} & c1: \textbf{while (TRUE)} \\
\quad \{ \quad \textbf{await } p = c\,; & \quad \{ \quad \textbf{await } p \neq c\,; \\
\quad p2:\ box := Put(box)\,; & \quad c2:\ box := Get(box)\,; \\
\quad\quad p := p \oplus 1 & \quad\quad c := c \oplus 1 \\
\quad \} & \quad \}
\end{array}
$$

Run the translator and look at the translation. The first thing we notice is that the translation has added a variable pc to represent the control state. It defines *ProcSet* to equal $\{0\} \cup \{1\}$, which equals the set $\{0, 1\}$ of process ids. The initial predicate *Init* therefore states that the initial value of pc is

$$
[self \in \{0, 1\} \mapsto \text{CASE } self = 0 \rightarrow \text{``p1''}
$$
$$
\Box\ \ self = 1 \rightarrow \text{``c1''}]
$$

You can probably guess that *Init* asserts that pc equals a function with domain $\{0, 1\}$ such that $pc[0]$ equals "p1" and $pc[1]$ equals "c1", from which you can figure out the meaning of the CASE construct$^\Box$.

Let's have TLC display the beginning of a behavior of this algorithm. Add the formula $pc[1] =$ "c1" as an invariant to your model and run TLC. It should produce an error trace containing four states, the last state violating this "invariant" because it has $pc[1]$ equal to "c2". You should be able to figure out how TLC uses the operators @@ and :> to write functions□. (Clicking on the ⊞ next to pc in one of the states may help.) We can continue this behavior to:

$$
\begin{bmatrix}
p &= 0 \\
c &= 0 \\
box &= \langle\,\rangle \\
pc &= 0 :> \text{"p1"} \; @@ \\
& 1 :> \text{"c1"}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
p &= 0 \\
c &= 0 \\
box &= \langle\,\rangle \\
pc &= 0 :> \text{"p2"} \; @@ \\
& 1 :> \text{"c1"}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
p &= 1 \\
c &= 0 \\
box &= \langle \text{"widget"} \rangle \\
pc &= 0 :> \text{"p1"} \; @@ \\
& 1 :> \text{"c1"}
\end{bmatrix}
\rightarrow
$$

$$
\begin{bmatrix}
p &= 1 \\
c &= 0 \\
box &= \langle \text{"widget"} \rangle \\
pc &= 0 :> \text{"p1"} \; @@ \\
& 1 :> \text{"c2"}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
p &= 1 \\
c &= 1 \\
box &= \langle\,\rangle \\
pc &= 0 :> \text{"p1"} \; @@ \\
& 1 :> \text{"c1"}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
p &= 1 \\
c &= 1 \\
box &= \langle\,\rangle \\
pc &= 0 :> \text{"p2"} \; @@ \\
& 1 :> \text{"c1"}
\end{bmatrix}
\rightarrow
$$

$$
\begin{bmatrix}
p &= 0 \\
c &= 1 \\
box &= \langle \text{"widget"} \rangle \\
pc &= 0 :> \text{"p1"} \; @@ \\
& 1 :> \text{"c1"}
\end{bmatrix}
\rightarrow \cdots
$$

Letting this be the behavior $s_1 \rightarrow s_2 \rightarrow \cdots$, let's compute the corresponding behavior $\overline{s_1} \rightarrow \overline{s_2} \rightarrow \cdots$ of module *Alternate* defined by the refinement mapping $\overline{b} \triangleq p \oplus c$, $\overline{box} \triangleq box$. Using the same procedure as before (adding the values of $\overline{b}$ and $\overline{box}$ to each state, deleting the variables p, c, box, and pc of *Handshake*, and erasing the overbars from $\overline{b}$ and $\overline{box}$), we get this behavior:

$$
\begin{bmatrix}
b &= 0 \\
box &= \langle\,\rangle
\end{bmatrix}
\rightarrow
\begin{bmatrix}
b &= 0 \\
box &= \langle\,\rangle
\end{bmatrix}
\rightarrow
\begin{bmatrix}
b &= 1 \\
box &= \langle \text{"widget"} \rangle
\end{bmatrix}
\rightarrow
\begin{bmatrix}
b &= 1 \\
box &= \langle \text{"widget"} \rangle
\end{bmatrix}
\rightarrow
$$

$$
\begin{bmatrix}
b &= 0 \\
box &= \langle\,\rangle
\end{bmatrix}
\rightarrow
\begin{bmatrix}
b &= 0 \\
box &= \langle\,\rangle
\end{bmatrix}
\rightarrow
\begin{bmatrix}
b &= 1 \\
box &= \langle \text{"widget"} \rangle
\end{bmatrix}
\rightarrow \cdots
$$

This behavior has stuttering steps—steps that repeat the same state—that were not in the behaviors of algorithm *Alternation* that we have computed. So, it looks like this version of algorithm *Handshake*, with the additional labels, doesn't implement *Alternation* under the refinement mapping. Let's have TLC verify that it doesn't. Run TLC on the same model as before, except with $pc[1] =$ "c1" removed from the list of invariants. TLC reports no error! What's going on?

6.7.2 Temporal Logic and Stuttering

To understand why this version of algorithm *Handshake* still implements *Alternation* under the refinement mapping, we must examine the temporal formulas *Spec* that are the behavior specifications of our algorithms. Ignoring fairness, these specifications *Spec* have the form $Init \wedge \Box[Next]_{vars}$. A non-temporal formula like *Init* or *Next* is an assertion about a step. Formula *Next* is true of a step $s \rightarrow t$ iff it is true when we substitute for each unprimed variable v the value of v in state s, and for each primed variable v' the value of v in state t. Since *Init* has no primed variables, whether or not it is true on a step $s \rightarrow t$ depends only on s, not on t. We can therefore think of *Init* being true or false for a state.

We consider a formula without any temporal operators to be the temporal formula that is true of a behavior iff it is true of the first step of that behavior. Since the truth of *Init* in a step depends only on the step's first state, *Init* is the temporal formula that is true of a behavior iff it is true in the behavior's first state.

For a formula F with no temporal operators, $\Box F$ is the temporal formula that is true of a behavior iff it is true of *every* step of the behavior. Thus, the formula $Init \wedge \Box[Next]_{vars}$ is true of a behavior iff *Init* is true of the first state of the behavior and $[Next]_{vars}$ is true of every step in the behavior. For any action formula A and state expression e, the formula $[A]_e$ is defined to equal $A \vee \text{UNCHANGED } e$. The same reasoning used above for an ordered pair of variables shows that, if *vars* is a tuple of variables, then UNCHANGED *vars* is true of a step iff the step leaves all those variables unchanged. Thus, if *vars* is the tuple of all the specification's variables, a $[Next]_{vars}$ step is one that is either a *Next* step or else is a stuttering step—one that leaves all the specification's variables unchanged.

If we wanted an algorithm to disallow stuttering steps, we would have to write its specification as $Init \wedge \Box Next$. Try it. Change the translation's definition of *Spec* in module *Handshake* by replacing $\Box[Next]_{vars}$ with $\Box Next$, and save the module. You will get the parsing error:

> $\Box$ followed by action not of form $[A]_v$.

$\Box Next$ is not a legal TLA$^+$ formula. TLA$^+$ does not let us write a specification that disallows stuttering steps. (Run the translator to restore the original definition of *Spec*.)

We now see why the version of algorithm *Handshake* with the additional labels implements algorithm *Alternation* under our refinement mapping. The extra steps introduced by the additional labels are mapped by the refinement mapping to stuttering steps, which are allowed by the *Alternation* algorithm's behavioral specification.

Writing specifications so they allow stuttering steps will seem strange to

most readers. Why does TLA$^+$ force us to do it? The answer is that allowing stuttering steps yields the simplest, most natural definition of implementation. To understand why, we must examine more closely what a *state* is.

In describing our Standard Model, I wrote that a state is an assignment of values to variables. I didn't say *what* variables. In describing behaviors, I have described each state of the behavior by stating what values it assigns to the system's variables. This would naturally have led you to believe that a state is an assignment of values just to the system's variables, so what constitutes a state depends on the system under consideration.

I have misled you. In TLA$^+$, a state is an assignment of values to all of the (infinitely many) possible variables. Any formula can contain only a finite number of variables, so it describes only the values assigned to those variables. The *Alternation* specification mentions only the variables b and *box*. Hence, whether or not a behavior σ satisfies that specification depends only on the values assigned to b and *box* by the states of σ. Those states also assign values to all other possible TLA$^+$ variables: q, r, $a2\_4xyZ9muuP$, and so on. However, the values assigned to those other variables have no effect on whether or not σ satisfies the specification.

Think of a state as specifying a universe that might include the one-bit clock, the Die Hard system, Euclid's algorithm, and the Internet. A behavior satisfying the Die Hard specification is not a behavior of a system of buckets and water; rather, it is a behavior of the universe in which the part of the universe that describes how much water is in the buckets (the variables *big* and *small*) satisfies formula *Spec* of module *PDieHard*. A specification of the Die Hard system should not be violated because the one-bit clock ticked (changing the value of the variable b) between two successive steps of the Die Hard system. It isn't violated only because the *DieHard* specification allows steps in which the value of b and other variables change while the values of *big* and *small* remain the same. These are the stuttering steps (the steps satisfying UNCHANGED *vars*) allowed by formula *Spec*.

Similarly, whether a behavior τ should implement the *Alternation* specification under the refinement mapping $\overline{b} \triangleq p \oplus c$, $\overline{box} \triangleq box$ should depend only on the changes to the value assigned by the states of τ to $\overline{b}$ and $\overline{box}$. It shouldn't depend on whether or not the values assigned to *big* or *small* changed between changes to $\overline{b}$ and $\overline{box}$.

Any temporal formula we write should describe only the values of the variables in that formula. If a temporal formula F does not contain the one-bit clock's variable b, then whether or not F is true or false of a behavior σ should not depend on whether or not the one-bit clock ticked between changes to the variables of F. Whether or not F is true of a behavior σ should not be changed by adding or removing steps that do not change any variables of F. A formula F that has this property is said to be *insensitive to stuttering*.

The syntax of TLA$^+$ allows you to write only formulas that are insensitive

to stuttering. It doesn't allow you to write the formula $\Box Next$ because adding a stuttering step to a behavior satisfying this formula could make the formula false on that behavior.

I have lied to you by writing that a finite behavior represents a behavior in which the system halts. When a system halts, the values assigned to its variables stop changing; the entire universe doesn't come to a stop. Hence, an execution in which a system halts is naturally represented as a behavior that ends with an infinite sequence of states in which the system's variables remain the same— that is, an infinite sequence of stuttering steps of that system. We do not need behaviors containing only a finite number of states to represent a system that halts, so we define a behavior to be an infinite sequence of states. Thus, when a TLC error trace ends with <Stuttering>, it is describing a behavior in which the preceding state is repeated infinitely many times.

This confession calls for a re-examination of fairness. Recall that in , I wrote that weak fairness of an action A is satisfied by a behavior $s_1 \rightarrow s_2 \rightarrow \cdots$ iff it satisfies these two conditions:

3a. The behavior does not end in a state s_n in which A is enabled.

3b. If the behavior is infinite, then there is no n such that the infinite behavior $s_n \rightarrow s_{n+1} \rightarrow$ has no A steps but A is enabled in all of its states.

As we now know, every behavior is infinite, and ending in a state s_n really means that all the steps in $s_n \rightarrow s_{n+1} \rightarrow \cdots$ are stuttering steps. We can therefore combine these conditions into the single condition:

There is no n such that $s_n \rightarrow s_{n+1} \rightarrow \cdots$ has no A steps but A is enabled in all of its states.

There is a problem with this condition. If A allows stuttering steps, then the condition is not insensitive to stuttering. Adding stuttering A steps from a behavior can make the condition become true.

To solve this problem, we define $\langle A \rangle_{vars}$ to equal $A \wedge (vars' \neq vars)$. In other words, an $\langle A \rangle_{vars}$ step is an A step that changes $vars$. If $vars$ is the tuple of all system variables, then a step that leaves $vars$ unchanged is a stuttering step. Therefore an $\langle A \rangle_{vars}$ step is a non-stuttering A step. We then define the weak fairness formula $\text{WF}_{vars}(A)$ by:

$\text{WF}_{vars}(A)$ is satisfied by a behavior $s_1 \rightarrow s_2 \rightarrow \cdots$ iff there is no n such that $s_n \rightarrow s_{n+1} \rightarrow \cdots$ has no $\langle A \rangle_{vars}$ step but $\langle A \rangle_{vars}$ is enabled in all of its states.

Weak fairness is an important concept, so you should make sure you understand this definition. Observe that it is equivalent to:

$\mathrm{WF}_{vars}(A)$ asserts of a behavior $s_1 \to s_2 \to \cdots$ that if $\langle A \rangle_{vars}$ is enabled in all states s_m with $m \geq n$ for some n, then there is an $m \geq n$ such that $s_m \to s_{m+1}$ is an $\langle A \rangle_{vars}$ step.

This can be expressed informally as: $\mathrm{WF}_{vars}(A)$ asserts that if $\langle A \rangle_{vars}$ ever becomes enabled forever, then an $\langle A \rangle_{vars}$ step must eventually occur. A common situation is one in which an $\langle A \rangle_{vars}$ step disables $\langle A \rangle_{vars}$—for example, if A is the *Tick* or *Tock* action of algorithm *TickTock*. In this case, weak fairness of $\langle A \rangle_{vars}$ implies that the action can never be enabled forever, since if it were enabled forever, then weak fairness would imply that an $\langle A \rangle_{vars}$ action must occur, implying that it can't be enabled forever. Some people find this observation confusing. If you're one of them, you should re-read it until this all becomes obvious rather than confusing.

Most of the time, you can forget that a state assigns values to all variables and think only of a *system state*, which is an assignment of values to the system's variables. We will continue to describe behaviors by describing only the system state, ignoring all the irrelevant variables. When thinking about a system implementing a specification, it's important to remember that a specification allows stuttering steps. However, we usually ignore stuttering steps when considering a system in isolation.

For most weak fairness formulas $\mathrm{WF}_{vars}(A)$ that occur in specifications, action A does not allow stuttering steps that start in a reachable state. In that case, we can forget about the distinction between A and $\langle A \rangle_{vars}$, and I will usually consider weak fairness of A to be the assertion that is true of a behavior iff it doesn't end in a sequence of states in which A is always enabled but no A step occurs.

6.7.3 A Finer-Grained Algorithm

Let's add still more steps to the algorithm by putting the label $p3$ on the producer's $p := p \oplus 1$ statement and the label $c3$ on the consumer's $c := c \oplus 1$ statement. When we add steps to an algorithm by adding labels, we say that the resulting algorithm is *finer-grained* than the original, and that the original is *coarser-grained* than the new algorithm.

Run the translator and run TLC to see if this finer-grained algorithm *Handshake* still implements *Alternation* under our refinement mapping. TLC reports the error:

```
Action property line 50, col 20 to line 50, col 32 of module Alternation
is violated.
```

Clicking on the location in the error message shows that the error is in the subformula $\square[Next]_{vars}$ of module *Alternation*. The error trace displayed by TLC is the beginning of a behavior σ of algorithm *Handshake* for which the behavior $\overline{\sigma}$ is not a behavior of algorithm *Alternation* because it contains a step

that is not a $[Next]_{vars}$ step. Since TLC reports the shortest possible error trace for a violation of a safety property, the problem must be in the last step of that trace.

As you did before, use TLC's Trace Explorer to compute the values of $p \oplus c$ (which equals $\overline{b}$) for the states in the trace. You will see that the last step of the error trace is one in which box (which equals $\overline{box}$) changes, but $\overline{b}$ is not changed. However, in algorithm $Alternation$, box and b always change together. Hence the corresponding change to the variables box and b of module $Alternation$ is not a $[Next]_{vars}$ step (for $Next$ and var defined in that module).

For the finer-grained $Handshake$ algorithm to implement alternation, it needs to implement algorithm $Alternation$ under some refinement mapping for which $\overline{box}$ equals box. It doesn't have to be the refinement mapping we have been using. To find such a refinement mapping, we must define $\overline{b}$ so it changes when box changes. We do that by defining $\overline{b}$ to equal $vp \oplus vc$, where:

> vp has the same value as p except that it's changed by the execution of statement $p2$ and left unchanged by execution of $p3$.

> vc has the same value as c except it's changed by the execution of statement $c2$ and left unchanged by execution of $c3$.

(The v stands for *virtual*.) We define vp to be that state function that equals p except when the producer is at control point $p3$, when it equals $p \oplus 1$; and we define vc similarly. The definitions are:

$$vp \triangleq \text{IF } pc[0] = \text{"p3" THEN } p \oplus 1 \text{ ELSE } p$$
$$vc \triangleq \text{IF } pc[1] = \text{"c3" THEN } c \oplus 1 \text{ ELSE } c$$

```
vp == IF pc[0] = "p3" THEN p (+) 1 ELSE p
vc == IF pc[1] = "c3" THEN c (+) 1 ELSE c
```

Add these definitions to the module and add the statement

$$A2 \triangleq \text{INSTANCE } Alternation \text{ WITH } b \leftarrow vp \oplus vc$$

Have TLC check property $A2!Spec$. (Don't forget to remove or uncheck the property $A!Spec$.) This time, TLC does not report violation of an action property. However, it does report the error:

```
Temporal properties were violated.
```

(If it doesn't, make sure that the two algorithms specify fairness of the consumer processes and not the producer processes.)

The error trace shows a behavior σ that stops (ends in an infinite sequence of stuttering steps) with the producer at $p3$. This behavior is allowed because there is no fairness requirement for the producer, so it can stop taking steps, and the consumer can take no step (its next-state action is not enabled) because it is at $c1$ and $p \neq c$ equals FALSE. However, in the behavior $\overline{\sigma}$ corresponding to σ under the refinement mapping, the producer has performed its $p1$ action and the

consumer's next-state action is enabled, but the consumer does nothing. Hence, the consumer's fairness requirement is not satisfied by $\overline{\sigma}$, so σ is a behavior of (the finer-grained) algorithm *Handshake* but $\overline{\sigma}$ is not a behavior of algorithm *Alternation*.

To make the the finer-grained *Handshake* algorithm implement alternation, we must add a fairness requirement on the producer that prevents it from stopping with control at $p3$. We can do this by requiring weak fairness of action $p3$—that is, by adding the conjunct $\mathrm{WF}_{vars}(p3)$ to the definition of *Spec*. To specify this requirement in the PlusCal code, we specify weak fairness of the producer process except for its $p1$ and $p2$ actions. We exempt an action from a fairness requirement by putting a dash (-) after its label. (Spaces before and after the ":" are allowed but not required.) So, we change the producer process to:

```
fair process ( Producer = 0 )
  { p1:- while ( TRUE )
          {      await p = c ;
            p2:- box := Put(box) ;
                 p3: p := p ⊕ 1
          }
  }
```

This causes the translation to add the following conjunct to the definition of *Next*:

$$\mathrm{WF}_{vars}((pc[0] \notin \{\text{"p1"}, \text{"p2"}\}) \wedge Producer)$$

This is equivalent to $\mathrm{WF}_{vars}(p3)$ because the definitions of *Producer*, $p1$, $p2$, and $p3$ imply that $(pc[0] \notin \{\text{"p1"}, \text{"p2"}\}) \wedge Producer$ is equivalent to $p3$. TLC verifies that, with this additional fairness requirement, algorithm *Handshake* implements algorithm *Alternation* under the refinement mapping.

6.8   Temporal Logic and Refinement

We now examine what refinement means in terms of temporal logic. Let's start by considering what it means for a system to satisfy a property, beginning with the simple property of invariance of a state predicate. Remember that a temporal formula F is an assertion about behaviors, meaning that it is true or false of a behavior. We often say that a behavior σ *satisfies* F if F is true of σ. A temporal formula that is satisfied by all behaviors is called a *theorem*, or simply a *true formula*.

Theorems versus provable formulas.

A state predicate I is an invariant of a specification *Spec* iff it is true in all states of every behavior σ that satisfies *Spec*. The temporal formula $\Box I$ is true of σ iff I is true in all states of σ. Thus, I is an invariant of *Spec* iff, for any behavior σ, if σ satisfies *Spec* then σ satisfies $\Box I$. This is the case iff every

behavior σ satisfies the formula $Spec \Rightarrow \Box I$. In other words, I is an invariant of $Spec$ iff the formula $Spec \Rightarrow \Box I$ is a theorem.

In general, a property P is a temporal formula. A specification $Spec$ is said to *satisfy* P iff $Spec \Rightarrow P$ is a theorem. Invariance of a state predicate I means that $Spec$ satisfies the property $\Box I$.

Now let's consider implementation under a refinement mapping. For concreteness, let's consider the implementation of algorithm *Alternation* by algorithm *Handshake* of Section 6.7.1 under the refinement mapping $\overline{b} \triangleq p \oplus c$, $\overline{box} \triangleq box$. Using the same name for different formulas gets confusing, so let's use subscripts to indicate in which specification an expression is defined. For example, let $Init_A$ be the formula $Init$ defined in the *Alternation* spec, and let $vars_H$ be expression $vars$ defined in the *Handshake* specification.

Now that I've confessed that a state is an assignment of values to *all* variables, I should restate what it means for algorithm *Handshake* to implement algorithm *Alternation* under the refinement mapping—without talking about states of a system. For any state s, we define $\overline{s}$ to be some state that assigns the value $\overline{b}$ to the variable b and the value $\overline{box}$ to the variable box. It doesn't matter what values $\overline{s}$ assigns to other variables. For any behavior $s_1 \to s_2 \to \cdots$, we define $\overline{s_1 \to s_2 \to \cdots}$ to be the behavior $\overline{s_1} \to \overline{s_2} \to \cdots$. Implementation under the refinement mapping means that if σ is any behavior satisfying $Spec_H$, then the behavior $\overline{\sigma}$ satisfies $Spec_A$.

For any formula F_A defined in the *Alternation* spec, define $\overline{F_A}$ to be the formula obtained from F_A by substituting $\overline{b}$ for b and $\overline{box}$ for box. (Of course, the latter substitution does nothing because $\overline{box}$ equals box for this particular refinement mapping.) For example, we have

$$Init_A \triangleq \land\ b = 0 \qquad\qquad \overline{Init_A} \triangleq \land\ (p \oplus c) = 0$$
$$\land\ box = \langle\rangle \qquad\qquad\qquad\qquad \land\ box = \langle\rangle$$

$$Consumer_A \triangleq \land\ b = 1 \qquad\qquad \overline{Consumer_A} \triangleq \land\ p \oplus c = 1$$
$$\land\ box' = Tail(box) \qquad\qquad\qquad \land\ box' = Tail(box)$$
$$\land\ b' = 0 \qquad\qquad\qquad\qquad\qquad \land\ (p \oplus c)' = 0$$

where the definition of Get_A has been expanded in the definition of $Consumer_A$. For any step $s \to t$:

> $Consumer_A$ is true on the step $\overline{s} \to \overline{t}$

> iff $Consumer_A$ is true with b and box replaced by their values in state $\overline{s}$ and b' and box' replaced by the values of b and box in state $\overline{t}$
> [by definition of what it means for an action to be true on a step]

> iff $Consumer_A$ is true with b and box replaced by the values of $\overline{b}$ and $\overline{box}$ in state s and b' and box' replaced by the values of $\overline{b}$ and $\overline{box}$ in state t
> [by definition of $\overline{s}$ and $\overline{t}$]

iff $\overline{Consumer_A}$ is true on the step $s \to t$.

[by definition of what it means for an action to be true on a step]

It's clear that this is a general result for any action or state predicate. In particular $Next_A$ is true on $\overline{s} \to \overline{t}$ iff $\overline{Next_A}$ is true on $s \to t$, and $Init_A$ is true on a state $\overline{s}$ iff $\overline{Init_A}$ is true on s.

The analogous result holds for temporal formulas. In particular, if σ is any behavior, $\overline{\sigma}$ satisfies $Spec_A$ iff σ satisfies $\overline{Spec_A}$. Remember that implementation under the refinement mapping means that for any behavior σ, if σ satisfies $Spec_H$, then $\overline{\sigma}$ satisfies $Spec_A$. Thus, it means that for any behavior σ, if σ satisfies $Spec_H$, then σ satisfies $\overline{Spec_A}$. In other words, algorithm *Handshake* implements algorithm *Alternation* under the refinement mapping means that $Spec_H$ satisfies the property $\overline{Spec_A}$, which means that $Spec_H \Rightarrow \overline{Spec_A}$ is a theorem.

How do we prove the theorem $Spec_H \Rightarrow \overline{Spec_A}$? The definitions of $Spec_H$ and $Spec_A$ are:

$$Spec_H \triangleq Init_H \wedge \Box[Next_H]_{vars_H} \wedge \text{WF}_{vars_H}(Consumer_H)$$
$$Spec_A \triangleq Init_A \wedge \Box[Next_A]_{vars_A} \wedge \text{WF}_{vars_A}(Consumer_A)$$

Since overbarring a formula means substituting expressions for variables in the formula, it's clear that $\overline{Spec_A}$ equals

$$\overline{Init_A} \wedge \Box\overline{[Next_A]}_{vars_A} \wedge \overline{\text{WF}_{vars_A}(Consumer_A)}$$

From this, it follows that to prove $Spec_H \Rightarrow \overline{Spec_A}$, it suffices to prove:

1. $Spec_H \Rightarrow \overline{Init_A}$

2. $Spec_H \Rightarrow \Box\overline{[Next_A]}_{vars_A}$

3. $Spec_H \Rightarrow \overline{\text{WF}_{vars_A}(Consumer_A)}$

We prove 1 by proving:

R1. $Init_H \Rightarrow \overline{Init_A}$

This follows easily from the definition of $\overline{Init_A}$ given above and the definition of $Init_H$ produced by the translator, since $0 \oplus 0$ equals 0.

The obvious way to prove formula 2 is to prove

$2a.$ $\Box[Next_H]_{vars_H} \Rightarrow \Box\overline{[Next_A]}_{vars_A}$

whose truth asserts that, for every behavior σ, if every step of σ is a $[Next_H]_{vars_H}$ step, then every step of σ is a $\overline{[Next_A]}_{vars_A}$ step. This is true if it is true that every $[Next_H]_{vars_H}$ is a $\overline{[Next_A]}_{vars_A}$ step. In other words, we can prove $2a$ by proving:

2b. $[Next_H]_{vars_H} \Rightarrow [\overline{Next_A}]_{\overline{vars_A}}$

Because $[Next_H]_{vars_H}$ equals $Next_H \vee \text{UNCHANGED } vars_H$, to prove 2b it suffices to prove:

2b1. $Next_H \Rightarrow [\overline{Next_A}]_{\overline{vars_A}}$

2b2. $\text{UNCHANGED } vars_H \Rightarrow [\overline{Next_A}]_{\overline{vars_A}}$

Condition 2b2 is obviously true, since $[\overline{Next_A}]_{\overline{vars_A}}$ equals $[\overline{Next_A}] \vee$ $\text{UNCHANGED } \overline{vars_A}$, and $\text{UNCHANGED } vars_H$ implies $\text{UNCHANGED } \overline{vars_A}$ because $\text{UNCHANGED } vars_H$ asserts that all the variables of specification *Handshake* are unchanged, which implies that $\overline{v}$ is unchanged for all the (two) variables v of *Alternation*. So, we just have to prove 2b1.

To prove 2b1, we must prove that it is true for all steps $s \to t$—even ones in s and t assign non-numerical values to p and c states. This is impossible—for example, we don't know what $p \oplus 1$ equals if p equals "abc". To prove 2, it suffices to prove that formula 2b1 is true for all steps $s \to t$ in which s and t are reachable states of $Spec_H$—that is, states that can occur in a behavior satisfying $Spec_H$. Any invariant of $Spec_H$ is true in all reachable states. Therefore, to prove that 2b1 is true for steps $s \to t$ where s and t are reachable, it suffices to prove:

R2. $Inv_H \wedge Inv_H{'} \wedge Next_H \Rightarrow [\overline{Next_A}]_{\overline{vars_A}}$ where $Spec_H \Rightarrow \Box Inv_H$ is true.

For our example, the invariant we need is

$$Inv_H \triangleq \begin{aligned} &\wedge\ p \in \{0,1\} \\ &\wedge\ c \in \{0,1\} \\ &\wedge\ (pc[0] = \text{"p2"}) \Rightarrow (p = c) \\ &\wedge\ (pc[1] = \text{"c2"}) \Rightarrow (p \neq c) \end{aligned}$$

Here is the proof of R2.

Writing proofs of liveness properties will become easier when you learn a little more temporal logic. I will therefore defer an explanation of how to prove condition 3.

Question 6.6 Show that the formula Inv_H defined above is an invariant of the *Handshake* specification. Why isn't it an inductive invariant? How can we strengthen it to obtain an inductive invariant?   ANSWER

Let's now return to the INSTANCE statement

$$A \triangleq \text{INSTANCE } Alternation \text{ WITH } b \leftarrow p \oplus c,\ box \leftarrow box$$

and the formula $A!Spec$ that we added to the Properties subsection of the TLC model. For every symbol F defined in module *Alternation*, this statement defines $A!F$ to be what we have been writing $\overline{F}$. Thus, $A!Spec$ equals what we have   What does "!" mean?

been writing $\overline{Spec_A}$. Making $A!Spec$ a property of the model to be checked tells TLC to check that the formula $Spec \Rightarrow A!Spec$ is true, where $Spec$ is the specification selected by the model's What is the behavior spec? section. Thus, telling TLC to check the property $\Box I$, where I is a state predicate, tells it to check that $Spec \Rightarrow \Box I$ is true. It is equivalent to adding I to the Invariants section of the model.

If F is defined in module *Alternation* to be an operator with the definition:

$$F(p_1, \ldots, p_n) \;\triangleq\; exp$$

then $A!F$ is defined in module *Handshake* by

$$A!F(p_1, \ldots, p_n) \;\triangleq\; \overline{exp}$$

where $\overline{exp}$ is the expression obtained from exp by replacing each variable v of modle *Alternation* with $\overline{v}$. For example, if module *Alternation* contained the definition:

$$ProCon(i) \;\triangleq\; \begin{aligned}&\land\; b = i\\&\land\; b' = i \oplus 1\end{aligned}$$

then the INSTANCE statement would effectively define

$$A!ProCon(i) \;\triangleq\; \begin{aligned}&\land\; p \oplus c = i\\&\land\; (p \oplus c)' = i \oplus 1\end{aligned}$$

in module *Handshake*. (Of course, we can't actually write that definition because $A!ProCon$ is not a legal identifier.) Note that because the definitions of *Put* and *Get* in module *Alternation* contain no variables, the INSTANCE statement defines $A!Put$ and $A!Get$ in module *Handshake* to be the same as the operators *Put* and *Get* defined in that model.

Everything we have done here generalizes to proving that any safety specification (one with no fairness conditions) implements another safety specification under a refinement mapping. Let me restate it:

> To prove that $Init_H \land \Box[Next_H]_{vars_H}$ implements $Init_A \land \Box[Next_A]_{vars_A}$ under a refinement mapping, we prove
>
> $$Init_H \land \Box[Next_H]_{vars_H} \;\Rightarrow\; \overline{Init_A \land \Box[Next_A]_{vars_A}}$$
>
> by finding an invariant Inv_H of $Init_H \land \Box[Next_H]_{vars_H}$ (a formula for which $Init_H \land \Box[Next_H]_{vars_H} \Rightarrow \Box Inv_H$ is a theorem) and proving:
>
> R1. $Init_H \Rightarrow \overline{Init_A}$
> R2. $Inv_H \land Inv_H{}' \land Next_H \;\Rightarrow\; \overline{[Next_A]}_{\overline{vars_A}}$

Condition R2 is called *step simulation*. It asserts that any $Next_H$ step simulates, under the refinement mapping, either a $Next_A$ step or a stuttering step.

To prove that Inv_H is an invariant, we have to find an inductive invariant that implies it. We usually take Inv_H to be that inductive invariant.

6.9 Alternation Revisited

The main purpose of this discussion of alternation and the two-phase handshake protocol has been to introduce the concepts of refinement and insensitivity to stuttering. Now that you understand those concepts, we can describe the alternation problem in a somewhat more conventional way. Such syncronization problems are usually expressed in terms of pseudo-code. We will do it in terms of PlusCal.

Let's ignore variable declarations and consider only the two processes. We are given two arbitrary pieces of PlusCal code, *put* and *get*, which may contain labels, but can exit only by "falling off the bottom". They must be executed so that first *put* is executed, then *get* is executed, then *put* is executed, and so on. We must achieve this by implementing code sections *p_enter*, *p_exit*, *c_enter* and *c_exit*, using no variables that appear in *put* or *get*, in these two processes:

> PlusCal has a **goto** statement that can jump out from the middle of a piece of code.

```
process ( Producer = 0 )                    process ( Consumer = 1 )
  { pe: while ( TRUE )                         { ce: while ( TRUE )
        {      p_enter ;                             {      c_enter ;
        p:    put ;                                  g:    get ;
        px:   p_exit       }                         cx:   c_exit         }
  }                                            }
```

Let's call this generic two-process algorithm *AltImpl*.

A trivial solution is the algorithm containing the following processes, where we assume that the variable b does not occur in *put* or *get*.

```
process ( Producer = 0 )                    process ( Consumer = 1 )
  { pe: while ( TRUE )                         { ce: while ( TRUE )
        {      await b = 0 ;                          {      await b = 1 ;
        p:    put ;                                  g:    get ;
        px:   b := 1       }                         cx:   b := 0         }
  }                                            }
```

We can take this algorithm, which we call *AltSpec*, to be the specification of the alternation problem. An algorithm *AltImpl* is an alternation solution if it implements *AltSpec* under a refinement mapping with the following properties:

1. $\overline{v}$ equals v for every variable v that occurs in *put* or *get*.

2. $\overline{pc}$ satisfies:

   - $\overline{pc}[0]$ equals "pe" if $pc[0]$ equals "pe" or one of the label names in *p_enter*.
   - $\overline{pc}[0]$ equals $pc[0]$ if $pc[0]$ equals "p" or one of the label names in *put*.
   - $\overline{pc}[0]$ equals "px" if $pc[0]$ equals "px" or one of the label names in *p_exit*.

   and the analogous three conditions for $\overline{pc}[1]$.

We want the solution *AltImpl* to work for any *put* and *get* code. It's easy to see that this will be the case if the solution works when *put* and *get* are just **skip** statements, where a **skip** statement leaves all variabes (except *pc*) unchanged. We simplify the problem by replacing *put* and *get* with **skip** statements. The resulting algorithms are abstractions in which an execution of *put* or *get* is represented as a sequence of stuttering steps followed by execution of the **skip** (which modifies *pc*).

Our specification of alternation is then this algorithm *AltSpec*, in which we have changed the labels *p* and *g* to *put* and *get*. Create a new specification *AltSpec* and add the ASCII version of the algorithm to it. Run the translator and observe the translation of the **skip** statements. A solution to the alternation problem is a two-process algorithm with these processes

```
process ( Producer = 0 )                    process ( Consumer = 1 )
  { pe: while ( TRUE )                         { ce: while ( TRUE )
          {      p_enter ;                              {      c_enter ;
            put:  skip ;                           get:  skip ;
            px:   p_exit     }                     cx:   c_exit        }
  }                                            }
```

that implements algorithm *AltSpec* under a refinement mapping satisfying the following conditions:

- $\overline{pc}[0]$ equals "pe" if $pc[0]$ equals "pe" or one of the label names in *p_enter*.

- $\overline{pc}[0]$ equals "put" if $pc[0]$ equals "put".

- $\overline{pc}[0]$ equals "px" if $pc[0]$ equals "px" or one of the label names in *p_exit*.

and the corresponding three conditions for $\overline{pc}[1]$.

Question 6.7 Rewrite the two-phase handshake algorithm (by modifying its ASCII version) so it is a solution to the new statement of the alternation problem. Use TLC to check that it implements algorithm *AltSpec* under the appropriate refinement mapping.   ANSWER

Let's make the two-phase handshake example a little more interesting by writing a finer-grained version that splits the evaluation of each **await** test into two separate actions: reading the other process's variable and then testing its value. This is done in the following code, which introduces two features of PlusCal: **goto** statements and process-local variables. If you don't already know what a **goto** statement does, its TLA$^+$ translation will explain it.

$$
\begin{array}{ll}
\textbf{process (} Producer = 0 \textbf{)} & \textbf{process (} Consumer = 1 \textbf{)} \\
\quad \textbf{variable } tp = 0\,; & \quad \textbf{variable } tc = 0\,; \\
\quad \{\ pe:\ \textbf{while (} \text{TRUE} \textbf{)} & \quad \{\ ce:\ \textbf{while (} \text{TRUE} \textbf{)} \\
\qquad\quad \{\qquad tp := c\,; & \qquad\quad \{\qquad tc := p\,; \\
\qquad\quad pe1:\ \textbf{if (} p \neq tp \textbf{) \{ goto } pe \textbf{ \} }\,; & \qquad\quad ce1:\ \textbf{if (} c = tc \textbf{) \{ goto } ce \textbf{ \} }\,; \\
\qquad\quad put:\ \textbf{skip}\,; & \qquad\quad get:\ \textbf{skip}\,; \\
\qquad\quad px:\ \ p := p \oplus 1\quad \} & \qquad\quad cx:\ \ c := c \oplus 1\quad \} \\
\quad \} & \quad \}
\end{array}
$$

We could use the same name for the two process's local variables, but one of them would be renamed in the TLA$^+$ translation. The initial values of the local variables tp and tc are irrelevant; we'll see later what happens if we don't specify initial values for them.

We now must define the refinement mapping to show that the algorithm with these process declarations is an alternation solution. The refinement mapping again defines $\overline{b}$ to equal $p \oplus c$. The definition of $\overline{pc}$ follows directly from the conditions that it must satisfy. Read and understand the following definition, which defines $pcBar$ to equal $\overline{pc}$. (It is written with the CASE construct$^\square$ instead of with IF / THEN / ELSE to make it more obviously symmetric in the process id.)

$$
\begin{aligned}
pcBar \ \triangleq\ [i \in \{0, 1\} \mapsto\ &\text{CASE}\ \ i = 0 \to \text{IF }\ pc[0] = \text{``pe1''}\ \text{THEN}\ \text{``pe''} \\
&\qquad\qquad\qquad\qquad\qquad\quad\text{ELSE}\ \ pc[0] \\
\square\ \ \ &\ i = i \to \text{IF }\ pc[1] = \text{``ce1''}\ \text{THEN}\ \text{``ce''} \\
&\qquad\qquad\qquad\qquad\qquad\quad\text{ELSE}\ \ pc[1]]
\end{aligned}
$$

ASCII version

Question 6.8 Modify the module you wrote as the answer to Question 6.7 to have these finer-grained processes, and add this definition of $pcBar$ to it. Add an INSTANCE statement that instantiates module $AltSpec$ under the refinement mapping $b \leftarrow p \oplus c$, $pc \leftarrow pcBar$ and have TLC check that the specification implements $AltSpec$ under this refinement mapping.

Remove the initialization of the process local variables tp and tc (by removing the "= 0"s) and rerun the translator. Observe that the translation now asserts that tp and tc equal the declared constant $defaultInitValue$. For TLC to check a TLA$^+$ specification, the initial predicate must specify an initial value, or a set of possible initial values, for each variable. The translation uses the unspecified constant $defaultInitValue$ as the initial value of any variable whose initial value is not specified in its **variable** declaration. Now create a new TLC model of the specification. You will see that the model has set $defaultInitValue$ to be a model value.

Question 6.9 Suppose a PlusCal algorithm contains a global variable f for which there is an assignment statement of the form $f[e] := \ldots$. Why does the algorithm probably have to specify the initial value (or set of possible initial values) of f?

ANSWER

6.10   Round-Robin Synchronization

We now generalize from alternation to round-robin synchronization in which there are N processes numbered 0 through $N-1$, where each process i executes an operation $Op(i)$. An algorithm must ensure that these operations are executed in round-robin order:

$$Op(0) \rightarrow Op(1) \rightarrow \cdots \rightarrow Op(N-1) \rightarrow Op(0) \rightarrow Op(1) \rightarrow \cdots$$

To see how to state the more general problem, we first rewrite the one-bit clock algorithm.

6.10.1   The One-Bit Clock Revisited Again

In Section 6.2, we saw how to describe the one-bit clock as a two-process PlusCal algorithm. The module *TickTock* represented each process by a **process** declaration. We now write the same specification in a slightly different way, using a single process-set declaration to describe both processes.

   In the Toolbox, create a new specification named *TickTock*2, and add an EXTENDS *Integers* statement. Now add this PlusCal algorithm, which contains this **process** statement:

```
process ( TickTock ∈ {0, 1} )
    { t: while ( TRUE )
          { await b = self ;
            b := (self + 1) % 2 ;
          }
    }
```

It defines two processes having the process ids 0 and 1. The identifier *self* in the body of the process denotes the process's id. Thus, this **process** statement is equivalent to these two separate **process** statements:

```
process ( TickTock0 = 0 )               process ( TickTock1 = 1 )
    { t: while ( TRUE )                      { t: while ( TRUE )
          { await b = 0 ;                          { await b = 1 ;
            b := (0 + 1) % 2 ;                        b := (1 + 1) % 2
          }                                        }
    }                                        }
```

Since $(0+1) \% 2$ equals 1, and $(1+1) \%$ equals 0, these two processes *TickTock*0 and *TickTock*1 are the same as processes *Tick* and *Tock* of algorithm *TickTock*—except for the labels, which don't appear in the translations. A comparison of the TLA$^+$ translations of the two algorithms shows them to be equivalent.

6.10.2 An N-Valued Clock

Algorithm *TickTock2*, like the equivalent algorithm *TickTock*, describes a two-value clock. We now generalize it to describe an N-valued clock, for an arbitrary integer N greater than 1. Create a new specification named *ClockSpec* that begins with:

EXTENDS *Integers*	`EXTENDS Integers`
CONSTANT N	`CONSTANT N`
ASSUME $(N \in Nat) \wedge (N > 1)$	`ASSUME (N \in Nat) /\ (N > 1)`

We represent the clock by a variable c that, in an execution, has values lying in the set $0 \mathrel{..} (N-1)$; it initially equals 0. The PlusCal algorithm has the following **process** declaration, which is the obvious generalization of the one in algorithm *TickTock2* above.

```
process ( Tick ∈ 0 .. (N − 1) )
    { t: while ( TRUE )
            { await c = self ;
              c := (self + 1) % N
            }
    }
```

Add the ASCII text of the algorithm to the module and run the translator on it. Check that you haven't made any error by having TLC check that the algorithm satisfies the invariant $c \in 0 \mathrel{..} (N-1)$, for some value of N.

6.10.3 An Implementation of the N-Valued Clock

The heart of the handshake algorithm is an implementation of the one-bit clock under the refinement mapping $\overline{b} \triangleq (p + c) \% 2$, where the variables p and c have one-bit values that are each modified by a single process. We now generalize this to an implementation of the N-valued clock by an array ca of 1-bit values. To indicate how the implementation works, here are the values of ca with the corresponding values of $\overline{c}$ for the refinement mapping in the first few states of an execution, with $N = 4$.

	$ca[0]$	$ca[1]$	$ca[2]$	$ca[3]$	$\bar{c}$
State 1:	0	0	0	0	0
State 2:	1	0	0	0	1
State 3:	1	1	0	0	2
State 4:	1	1	1	0	3
State 5:	1	1	1	1	0
State 6:	0	1	1	1	1
State 7:	0	0	1	1	2
State 8:	0	0	0	1	3

$\vdots$

Think of the N processes with ids $0, \ldots, N-1$ arrayed clockwise around a circle like this, so process $(i+1) \% N$ follows process i. The value of $ca[i]$ is modified only by process i, and is read only by processes i and $(i+1) \% N$.

Open a new specification named *HSClock*, and have it extend the *Integers* module and declare the constant N to be an integer greater than 1 as in module *ClockSpec* above. Add the same definition of $\oplus$ as in the two-phase handshake algorithm:

$$a \oplus b \;\triangleq\; (a + b) \% 2 \qquad\qquad \texttt{a (+) b == (a + b) \% 2}$$

The *HSClock* algorithm initializes ca to be an array indexed by $0 \ldots (N-1)$ with $ca[i] = 0$ for all i in $0 \ldots (N-1)$.

variable $ca = [\,i \in 0 \ldots (N-1) \mapsto 0\,]$;

The code for process 0 differs from the code for processes $1, \ldots, N-1$.

process (*Proc0* $= 0$)
  { t: **while** (TRUE)
      { **await** $ca[0] \;=\; ca[N-1]$;
        $ca[0] \;:=\; ca[0] \oplus 1$
      }
  }

process (*Proc* $\in 1 \ldots (N-1)$)
  { t: **while** (TRUE)
      { **await** $ca[self] \;\neq\; ca[self-1]$;
        $ca[self] \;:=\; ca[self] \oplus 1$
      }
  }

Add the ASCII text of the algorithm to the module and run the translator. Examine the code and its translation. Observe that for $N = 2$, the values of

$ca[0]$ and $ca[1]$ behave the same as the values of p and c in the original *Handshake* algorithm (the one with only a single label in each process).

Type correctness of the algorithm is expressed by invariance of:

$$ca \in [0 \ldots (N-1) \to \{0, 1\}]$$ `ca \in [0..(N-1) -> {0, 1}]`

Have TLC check that this is indeed an invariant of the algorithm.

The algorithm executes the following sequence of actions: $Proc0$, $Proc(1)$, $Proc(2)$, ..., $Proc(N-1)$, $Proc0$, $Proc(1)$, ... It does this because, in every reachable state, either:

> HS1. $ca[0] = ca[1] = \ldots = ca[N-1]$, so only action $Proc0$ is enabled, or
>
> HS2. $ca[0] = \ldots = ca[i-1] \neq ca[i] = \ldots = ca[N-1]$ for some i in $1 \ldots (N-1)$, so only action $Proc(i)$ is enabled.

We expect *HSClock* to implement *ClockSpec* under a refinement mapping in which $Proc0$ implements $Tick(0)$ and $Proc(i)$ implements $Tick(i)$, for i in $1 \ldots (N-1)$. From HS1 and HS2, this leads us to define the refinement mapping by letting $\overline{c}$ equal

> IF $\exists i \in 1 \ldots (N-1) : ca[i] \neq ca[i-1]$
> THEN CHOOSE $i \in 1 \ldots (N-1) : ca[i] \neq ca[i-1]$
> ELSE 0

In module *HSClock*, define *cBar* to equal this expression and add:

ASCII version of definition and INSTANCE statement

$$CS \quad \triangleq \quad \text{INSTANCE } ClockSpec \text{ WITH } c \leftarrow cBar$$

(Note that there is an implicit WITH clause $N \leftarrow N$ that substitutes the declared constant N of module *HSClock* for the declared constant N of module *ClockSpec*.) Check that algorithm *HSClock* implements algorithm *ClockSpec* under this refinement mapping by having TLC check that specification *Spec* of *HSClock* satisfies the property $CS!Spec$, for a model with a small value of N.

Question 6.10 Prove that *HSClock* implements *ClockSpec* under this refinement mapping by proving conditions R1 and R2 for a suitable invariant *Inv*.   HINT

6.10.4 Round-Robin Synchronization

Question 6.11 (a) Add a **skip** statement and labels to algorithm *ClockSpec* to obtain a specification *RoundRobin* of round-robin synchronization that generalizes the specification of alternation by algorithm *AltSpec* we wrote in Section 6.9.

(b) Modify algorithm *HSClock* to obtain a generalization of the two-phase handshake protocol that implements specification *RoundRobin* under a suitable refinement mapping. Use TLC to check your answer.