The *Principles* and *Specification* Tracks

1 导论

1.1 并发计算

“并发的”即“同时发生的”。作为其名词形式，“并发”意味着多件事情同时发生。

“并发计算”是允许不同操作并发的一种计算形式。如今，大多数计算都是对现实世界里发生的动作的响应——比如用户移动或者点击了鼠标。现实世界里的并发动作意味着并发计算不可避免。当你移动鼠标的时候，你的计算机不能阻止你同时点击鼠标。

并行计算是一类特殊的并发计算。在并行计算中，为了加速完成任务，一个任务被分成多个子任务并发执行。原则上讲，并行是可以避免的，因为我们可以按序依次执行这些子任务。在实践中，并行或许不可避免，否则可能需要很长时间才能完成任务。但是，即使并行（同并发一样）不可避免，究其本质而言，它只是并发的一种简单形式。这是因为，在并行计算中，一件事情在何时发生是由我们——而非外部世界——所控制的。

1.2 为计算建模

如上所述，并发计算是允许不同操作并发的一种计算形式。但是，什么是计算呢？一个简单的答案是：计算就是一台计算机所做的那些事情。然而，由于种种原因，这个答案不能令人满意：

- 我们很难定义什么是计算机。一部手机是计算机吗？一部MP3 播放器呢？
- 如今，计算通常是由多台计算机组成的网络所完成的。
- 计算可以由非物理设备——尤其是程序与算法——完成。

一个更好的定义是：计算就是数字系统所做的那些事情。计算机、MP3 播放器、计算机网络、程序与算法都是数字系统。数字系统与其它系统的区别在于它的计算是由一组离散事件构成的。

袖珍计算器的计算是由诸如“按键”与“在屏幕上显示数字”这样的离散事件构成的，所以它是一种数字系统。不过，这些事件真的是离散事件吗？改变屏幕上显示的数字需要几毫秒，在这段时间内，屏幕由显示旧数字连续地变成显示新数字。计算器的使用者会认为这种变化是一个单一事件。但是，屏幕的设计者很可能不这么认为。当我们认为某个系统的计算是由离散事件构成的时候，对我们来说，这个系统就是一种数字系统。这时，我们直接说“该系统（如计算器）是一种数字系统”，而不说“我们认为该系统（如计算器）是一种数字系统”。更进一步，由于本书的主题是数字系统，我将省略“数字”这一定语并使用“系统”指代数字系统。

袖珍计算器系统的离散事件究竟是什么？从键盘输入数字3 是一个单一事件吗？还是说，按下按键3 与松开按键3 是两个不同的事件？计算器的使用者很可能会认为输入数字3 是一个单一事件。而对键盘的设计者来说，它又包含两个不同的事件。

本书的主题不是像计算器这样的物理系统，而是抽象系统——那些我们认为其计算是由离散事件构成的数字系统的抽象。我们研究的是抽象系统的原理。

对一个物理系统，我们如何决定该使用什么抽象呢？输入数字3 是一个事件还是包含了两个事件？这取决于抽象的目标。将输入某数字看作一个单一事件所得到的抽象比较简单。然而，它无法描述这样一种可能情况：先按下3，然后在松开3 之前按下4。键盘的设计者需要考虑这种可能性，因此，这种抽象不能满足他们的需求，他们需要区分“按键”与“松键”两种事件。尝试理解如何使用计算器的用户很可能并不关心同时按下两个键会发生什么，他们更喜欢依照简单抽象写成的使用手册。

所有学科都有这种关于抽象的问题。天文学家在研究行星运动时通常将行星抽象为一个质点。但是，如果需要考虑潮汐效应的话，这种抽象就不能满足要求了。

包含较少事件的抽象比较简单。包含较多事件的抽象则可以更精确地刻画实际系统。我们希望使用简单而又不失精确的抽象。虽然如何选择正确的抽象是一门艺术，但是还是有一些基本原则可供遵循。

选定了系统的某种抽象之后，我们需要决定如何表示这种抽象。系统的某种抽象的一种表示被称作该系统的一个模型。系统建模有多种方式。有的将事件看作原子对象。有的将状态——也就是变量的赋值——看作原子对象，而将事件定义为状态之间的转换。还有的将状态与事件名作为原子对象，此时，事件被定义为标有事件名的状态转换。还有一种建模方式将事件集合看作原子对象。不同的模型可以表达不同的性质，我们将会接触到多种模型。但是，我们用得最多的是被我们称为标准模型的如下模型：

> **标准模型** 一个抽象系统被描述为一组（系统）行为，其中每个（系统）行为代表系统的一次可能执行。一个（系统）行为是一个状态序列，而一个状态则是一种变量赋值。

在该模型中，事件，或者称步，被定义为系统行为中的状态转换。我发现上述标准模型是所有能够很好地扩展到复杂系统的模型中最简单的一种模型。

我们为系统的某种抽象建立模型。我们用*系统模型*（或者*某系统的某个模型*）表示为（数字）系统的某种抽象建立的模型。

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:

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

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

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

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..

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 $\land$ or $\lor$. We can therefore also write this formula as

$$\lor\ (b = 0)\ \land\ (b' = 1) \qquad \text{or} \qquad \lor \land\ b = 0 \quad \land\ b' = 1$$
$$\lor\ (b = 1)\ \land\ (b' = 0) \qquad\qquad\qquad\quad \lor \land\ b = 1 \quad \land\ b' = 0$$

We can also write the initial predicate as
$\lor\ b = 0$
$\lor\ b = 1$

Warning.

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

2.3 Writing the Specification

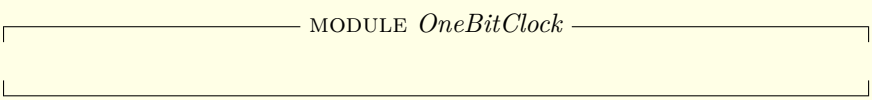
Let's now turn the initial predicate and next-state action into a TLA$^+$ specification. Open a new spec in the TLA$^+$ Toolbox. Name the specification and its root module *OneBitClock*. This creates a new module file named `OneBitClock.tla` and opens an editor on it.

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:



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 Init1 and Next1. These formulas are defined as follows.

$$\text{Init1} \triangleq (b = 0) \lor (b = 1)$$

ASCII version

$$\text{Next1} \triangleq \lor \land\ b = 0$$

$$\begin{array}{ll} & \land \, \mathrm{b}.2\mathrm{em} \; \backslash.' = 1 \\ \lor & \land \, \mathrm{b} = 1 \\ & \land \, \mathrm{b}.2\mathrm{em} \; \backslash.' = 0 \end{array}$$

Remember that you can click on the link to the ASCII version and copy the text.

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 $((\mathrm{b} = 0) \lor (\mathrm{b} = 1))$. 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.

The parsed icon in the lower-right corner tells you that the spec has no parsing errors.

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.

How to find help pages in the Toolbox. □

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

Does it do something else?

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 Init1 and Next1 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 Next1 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.

Name	Value
▲ <Initial predicate>	State (num = 1)
b	1
▲ <Action line 8, col 13 to line 9, col 25	State (num = 2)
b	"xyz"

This describes the following error trace:

b = 1 → b = "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:

▲ <Action line 8, col 13 to line 9, col 25	State (num = 2)

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 = “xyz”, TLC had to compute the value of the formula “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 “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□.

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.

$b \in \{0, 1\}$ `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.

$$\text{TypeOK} \triangleq \text{b} \in \{0, 1\}$$

```
TypeOK == b \in {0,1}
```

Save the spec and let's tidy up the model by using TypeOK rather than $\text{b} \in \{0, 1\}$ 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

$$(\text{b} = 0) \lor (\text{b} = 1)$$

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 $\text{b} = 1$.

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 $\text{b} = 1$. 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

$$\begin{array}{l} \lor \land \mathrm{b} = 0 \\ \quad \land \mathrm{b}' = 1 \\ \lor \land \mathrm{b} = 1 \\ \quad \land \mathrm{b}' = 0 \end{array}$$

We substitute 1 for b and simplify as follows.

$$\begin{array}{lll} \lor \land 1 = 0 & & \text{the formula obtained by substituting 1 for b in Next1.} \\ \quad \land \mathrm{b}' = 1 & & \\ \lor \land 1 = 1 & & \\ \quad \land \mathrm{b}' = 0 & & \\ = \quad \lor \land \textsc{false} & & \text{because } (0 = 1) = \textsc{false} \text{ and } (1 = 1) = \textsc{true} \\ \quad\quad \land \mathrm{b}' = 1 & & \\ \quad \lor \land \textsc{true} & & \\ \quad\quad \land \mathrm{b}' = 0 & & \\ = \quad \lor \textsc{false} & & \text{because } \textsc{false} \land \mathrm{F} = \textsc{false} \text{ and } \textsc{true} \land \mathrm{F} = \mathrm{F} \\ \quad \lor \mathrm{b}' = 0 & & \text{for any truth value F} \\ = \quad \mathrm{b}' = 0 & & \text{because } \textsc{false} \lor \mathrm{F} = \mathrm{F} \text{ for any truth value F.} \end{array}$$

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

$$\mathrm{b} = 1 \quad \rightarrow \quad \mathrm{b} = 0$$

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

$$\mathrm{b} = 1 \quad \rightarrow \quad \mathrm{b} = 0 \quad \rightarrow \quad \mathrm{b} = 1$$

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

$$\mathrm{b} = 1 \quad \rightarrow \quad \mathrm{b} = 0 \quad \rightarrow \quad \mathrm{b} = 1 \quad \rightarrow \quad \mathrm{b} = 0 \quad \rightarrow \quad \cdots$$

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

$$\mathrm{b} = 0 \quad \rightarrow \quad \mathrm{b} = 1 \quad \rightarrow \quad \mathrm{b} = 0 \quad \rightarrow \quad \mathrm{b} = 1 \quad \rightarrow \quad \cdots$$

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 underlined 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

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

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

$$(b = 0) \lor ((b = 1) \land (1 = b))$$

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

$$(b = 0) \lor ((b = 1) \land (2 = b))$$

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.□

The astute reader will have noticed that the two formulas Init1 and TypeOK, which equal $(b = 0) \lor (b = 1)$ 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:

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

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.

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

```
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:

Why is this formula true if b equals 42?

$$\text{TypeOK} \Rightarrow (\text{Next1} \equiv \text{Next2})$$

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 %□, 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.

$\text{Next3} \triangleq b' = (b + 1) \,\%\, 2$

```
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.

Where can an EXTENDS go?

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.2em \.′ = 1
ELSE b.2em \.′ = 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 $\mathrm{b} \in \{0, 1\};$ `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

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

$$\mathrm{Next} \triangleq \begin{array}{ll} \text{IF } \mathrm{b} = 0 \text{ THEN} & \mathrm{b}.2\mathrm{em}\ \backslash.' = 1 \\ \text{ELSE} & \mathrm{b}.2\mathrm{em}\ \backslash.' = 0 \end{array}$$

except that the translator formats them differently, inserting a comment and some unnecessary $\land$ 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 Init1 in module OneBitClock. We have seen the definition of Next above too, where we observed that it is equivalent to the definition of Next2 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 $\mathrm{b} := (\mathrm{b}+1) \,\%\, 2$, the translation will define Next to be the formula Next3 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.

?

←

→

C

I

S