

The *Principles* and *Specification* Tracks

1 导论

- 1.1 并发计算
- 1.2 为计算建模
- 1.3 Specification
- 1.4 Systems and Languages

2 单比特时钟

- 2.1 时钟的行为
- 2.2 描述行为
- 2.3 编写规约
- 2.4 以优良品质打印的规约
- 2.5 验证规约
- 2.6 Computing the Behaviors from the Specification
- 2.7 Other Ways of Writing the Behavior Specification
- 2.8 Specifying the Clock in PlusCal

?

←

→

C

I

S

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。

?

←

→

C

I

S

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.

?

←

→

C

I

S

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

?

←

→

C

I

S

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.

?

←

→

C

I

S

2 单比特时钟

我们考虑的第一个例子是一个尽可能简单的时钟：它交替显示“时间”0 与 1。这类时钟控制着你正用于阅读本书的计算机，它的“时间”表现为电缆上的电压。真实的时钟应该基本保持匀速行走。我们忽略这种需求，因为如果要描述它，我们还需要解释很多事情。这样一来，我们只需要考虑一个在两种状态之间交替的非常简单的计算设备：一个状态显示 0，另一个状态显示 1。

这似乎是个奇怪的例子，因为它不涉及并发。这个时钟一次只做一件事。在同一时刻，一个系统可以做任意多件事。一是任意多的简单的特殊情况，这是一个好的起点。学习使用 TLA⁺ 描述顺序系统能让我们掌握描述并发系统所需的大多数知识。

2.1 时钟的行为

我们使用标准模型建模时钟。也就是说，时钟的一次可能执行表示为一个行为。一个行为是一个状态序列，而一个状态则是对变量的一种赋值。我们用一个变量 b 表示“钟面”： b 赋值为 0 表示时钟显示 0，赋值为 1 表示时钟显示 1。我们用公式 $b = 0$ 表示“变量 b 赋值为 0”这一状态； $b = 1$ 的含义类似。

如果我们让时钟从 0 开始，那么它的行为可以形象地描述成：

$$b = 0 \rightarrow b = 1 \rightarrow b = 0 \rightarrow b = 1 \rightarrow \dots$$

其中，“...”表示时钟永不停歇。真实的时钟总会停下来；我们只能期望它运行的时间足够长。但是，考虑一个理想的永不停歇的时钟更为方便，而不必决定它究竟需要运行多长时间。因此，我们描述一个永不停歇的时钟。

我们也可以让时钟从 1 开始，在这种情况下，它的行为如下所示：

$$b = 1 \rightarrow b = 0 \rightarrow b = 1 \rightarrow b = 0 \rightarrow \dots$$

以上行为是单比特时钟仅有的两种行为。

需要记住的是，尽管我们一直在使用“时钟的行为”这一说法，实际上，我们所描述的是“真实时钟的某种抽象的标准模型的行为”。真实时钟的钟面从一个值连续地变换成下一个值。虽然，在数字时钟里，这种变换可能发生得太快，以至于我们看不到中间值，但是它们是客观存在的。在我们所描述的时钟抽象里，这种连续的变换被表示为离散的步骤（状态变化）。

2.2 描述行为

要描述一个计算设备，需要描述它的所有可能行为。我能够枚举单比特时钟的所有可能行为，但是枚举对于稍显复杂的计算设备就不可行了。有时，仅仅是展示复杂计算设备的单个行为都是困难的，更何况大多数计算设备的行为多到不胜枚举——通常是无穷多的。

?

←

→

C

I

S

FIXME: 跳出语法层面，我们发现，要描述计算设备，一门语言需要指明以下两点：

- 所有可能的初始状态。
- 所有可能的步骤。（步即状态转换。）

例如，使用某种（假想的）编程语言，单比特时钟可以描述如下：

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

第一行指明可能的初始状态是 $b = 0$ 或 $b = 1$ 。第二行指明如果 b 当前为 0，那么在下一个状态 b 为 1；如果 b 当前为 1，那么在下一个状态 b 为 0。

我们使用数学语言来描述初始状态与可能的后继状态，而不是发明一门全新的语言。我们使用布尔操作符 \wedge 与 \vee 。如果你不像熟悉算术操作符 $+$ 与 $-$ 那样熟悉这些简单的逻辑操作符，你应该[先看一下“关于逻辑的一些讨论”](#)。

初始状态易于描述。我们只需要断言 b 的初始值是 0 或者 1，用公式表示如下：

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

我们称该公式为初始谓词。

要描述可能的步骤，我们必须使用一个数学公式将 b 在两个状态——**FIXME: 该步骤的出发状态与后继状态**——中的值联系起来。我们使用 b 表示 b 在出发状态的值， b' 表示 b 在后继状态的值。有两种可能的步骤：一种是 $b = 0$ 且 $b' = 1$ ；另一种是 $b = 1$ 且 $b' = 0$ 。因此，所有可能的步骤可用如下公式表示：

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

由于括号的原因，即使这么短的公式也有点难以阅读。对于由合取与析取构成的更长的公式，我们几乎无法追踪其中的括号。TLA⁺ 允许我们将合取与析取写成标有符号 \wedge 或 \vee 的公式列表。因此，上式也可以写成：

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

我们也可以将初始谓词写成：

$$\begin{array}{l} \vee b = 0 \\ \vee b = 1 \end{array}$$

警告。

不管写成什么形式，我们通常将该公式称为次态动作，有时也称为次态关系。

2.3 编写规约

现在，我们将初始谓词与次态动作写入 TLA⁺ 规约。在 TLA⁺ 工具箱里[打开新规约](#)。将规约与它的根模块命名为 *OneBitClock*。这会创建一个名为 *OneBitClock.tla* 的模块文件，并打开一个编辑器。

编辑器中新模块的内容如下所示：


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

=====

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

第一行是开模块语句，最后一行是闭模块语句。开模块语句之前以及闭模块语句之后的内容都不属于模块，会被忽略。开模块语句中每个 - 序列与闭模块语句中的 = 序列的长度可以是大于 3 的任意值。开模块与闭模块显示如下：

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

现在，我们为上面定义的初始谓词与次态动作命名。习惯上，我称它们为 Init 与 Next。不过，由于后面我们会给出几种不同的定义，所以目前我们称之为 Init1 与 Next1，定义如下：

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

[ASCII version](#)

Next1 $\triangleq \begin{aligned} &\vee \wedge b = 0 \\ &\wedge b' = 1 \\ &\vee \wedge b = 1 \\ &\wedge b' = 0 \end{aligned}$

你可以点击上面的链接，查看并拷贝公式的 ASCII 版本。

上面两个 TLA+ 语句分别将 Init1 和 Next1 定义为一个公式。因此，在 Init1 定义语句之后的任何地方，Init1 都完全等价于 $((b = 0) \vee (b = 1))$ 。符号 \triangleq （输入 ==）读作“被定义为”。


保存模块，工具箱将自动解析。（否则，请查看[TLA+ 解析器配置菜单](#)。）解析器将报告六个错误，都在抱怨“b 是一个未知的操作符”。点击[解析错误](#)视图中的某条错误信息，将高亮显示该错误的位置，也就是 b 的某次特定出现。

模块中的每个符号都要么是一个原子 TLA+ 操作符，**FIXME: 要么在使用前已被定义过或声明过**。我们需要 声明 b 是一个变量。具体做法是在 Init1 定义语句之前插入如下声明语句：

VARIABLE b

VARIABLE b

保存模块，错误将自行消失。

右下角的  图标表明该规约没有解析错误。

当我们谈论单比特时钟的规约时，我们指的是

- 整个模块，或者
- 初始谓词与次态关系。

?
←
→
C
I
S

我们通常可以依靠上下文分辨出具体指代的是什么。为了避免混淆，当我们**FIXME: 意指前者时**，我们只谈论模块，而不谈规约。另外，我们使用术语行为规约指代后者。

2.4 以优良品质打印的规约

除了我们所编辑的 ASCII 版本的模块，工具箱还可以显示一种“以优良品质打印的”版本。这需要安装 `pdflatex` 程序。关于如何安装 `pdflatex`，以及如何配置工具箱的“优良打印”选项，请参考 工具箱的相关帮助页面。

如何找到工具箱的帮助页面。□

要生成模块的优良打印版本，请点击 File 菜单，选择 Produce PDF Version。生成的优良打印版本将显示在工具箱的一个单独的窗口中，其中的 TLA+ **FIXME: 表达式的显示方式与本书所展示的差不多**。你可以通过点击模块窗口左上角的 TLA Module 或者 PDF Viewer 标签在 ASCII 版本与优良打印版本之间进行切换。编辑 ASCII 版本并不会自动更新优良打印版本，你需要再次运行 File / Produce PDF Version 命令。

优良打印版本的文件名是 `OneBitClock.pdf`，与模块文件 `OneBitClock.tla` 存放在同一个文件夹下。你可以把它**FIXME: 打印成纸质版本**。

2.5 验证规约

现在，我们使用 TLC 模型验证器来验证该规约。[创建一个新的模型](#)。这将打开一个包含三个页面的模型编辑器，**FIXME: 默认打开的是 Model Overview 页面**。

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:

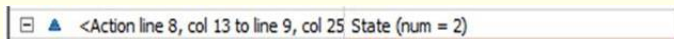
You can [resize the fields of the TLC Errors view](#).

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

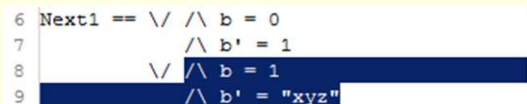
This describes the following error trace:

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

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 “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:

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

(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 .

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

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

?

←

→

C

I

S

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

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

We substitute 1 for b and simplify as follows.

$$\begin{aligned} & \vee \wedge 1 = 0 && \text{the formula obtained by substituting 1 for } b \text{ in } \text{Next1}. \\ & \quad \wedge b' = 1 \\ & \vee \wedge 1 = 1 \\ & \quad \wedge b' = 0 \\ & = \vee \wedge \text{FALSE} && \text{because } (0 = 1) = \text{FALSE} \text{ and } (1 = 1) = \text{TRUE} \\ & \quad \wedge b' = 1 \\ & \quad \vee \wedge \text{TRUE} \\ & \quad \wedge b' = 0 \\ & = \vee \text{FALSE} && \text{because } \text{FALSE} \wedge F = \text{FALSE} \text{ and } \text{TRUE} \wedge F = F \\ & \quad \vee b' = 0 && \text{for any truth value } F \\ & = b' = 0 && \text{because } \text{FALSE} \vee F = F \text{ for any truth value } F. \end{aligned}$$

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**

$$b = 1 \rightarrow b = 0 \rightarrow b = 1 \rightarrow b = 0 \rightarrow \dots$$

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

$$b = 0 \rightarrow b = 1 \rightarrow b = 0 \rightarrow b = 1 \rightarrow \dots$$

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

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

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 .

?

←

→

C

I

S

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

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' .

?

←

→

C

I

S

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.

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

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

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 :

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

Why is this formula true if b equals 42?

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.

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

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.

$$\begin{aligned} \text{IF } b = 0 \text{ THEN } b' &= 1 \\ \text{ELSE } b' &= 0 \end{aligned}$$

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 ∈ {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 IF b = 0 THEN b' = 1
 ELSE b' = 0`

except that the translator formats them differently, inserting a comment and some unnecessary `^` 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 `b := (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