

# The *Principles* and *Specification* Tracks

## 1 导论

- 1.1 并发计算
- 1.2 为计算建模
- 1.3 规约
- 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

## 3 Die Hard 问题

- 3.1 用 TLAPlus 描述 Die Hard 问题
- 3.2 Applying TLC
- 3.3 Expressing the Problem in PlusCal

## 4 Euclid 算法

- 4.1 The Greatest Common Divisor
  - 4.1.1 Divisors
  - 4.1.2 CHOOSE and the Maximum of a Set
  - 4.1.3 The GCD Operator
- 4.2 Comments
- 4.3 The Algorithm
- 4.4 The TLAPlus Translation
- 4.5 Checking Safety
- 4.6 Checking Liveness
- 4.7 The Translation Revisited
- 4.8 The Grain of Atomicity
- 4.9 Why Euclid's Algorithm Is Correct
  - 4.9.1 Proving Invariance
  - 4.9.2 Verifying  $GCD1-GCD3$
  - 4.9.3 Proving Termination
- 4.10 Euclid's Algorithm for Sets

?

←

→

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（或者称步骤）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 规约

规约是对系统模型的一种描述。形式化规约是用精确定义的语言写成的规约。我将使用术语系统规约（或者某系统的规约）表示一个系统模型的规约。

一个系统规约是该系统的某种抽象的某个模型的规约。它与实际系统相去甚远。我们为什么要写这样的规约呢？

一个规约就像一张建筑图纸。图纸与实际建筑也相去甚远，它是一张写了东西的纸，而一栋建筑则是由钢筋混凝土筑成的。我们无需解释为什么需要画建筑图纸。但是，值得指出的是，一张图纸之所以有用，在很大程度上正是因为它与它所勾画的实际建筑相去甚远。如果你想知道一栋建筑有多少平方英尺的办公面积，查看图纸比实际测量要来得容易。如果图纸是用支持自动计算的计算机程序绘制的，那就更容易了。

**FIXME:** 没有人会在建造一栋大型建筑之前不先绘制一张图纸的。我们也不应该在没有规约的情况下构造一个复杂的系统。人们会给出很多理由来说明为系统编写规约是在浪费时间：

- 无法从规约自动生成代码或者电路图。
- （经常）无法验证代码或者电路图是否正确地实现了某规约。
- 在构造系统时，我们会发现**FIXME: 一些需要改变系统需求的问题**。这导致规约没有描述实际系统。

**FIXME:** 当我们将这些理由看作关于“不用绘制图纸”的论证时，我们就能找到反驳它们的方法。

在筑造建筑物之前绘制的图纸是最有用的，它可以指导施工过程。然而，有的图纸是事后绘制的：比如，要改造一栋丢失了图纸的旧建筑。同样地，在构造系统之前编写的系统规约是最有用的。然而，有的规约也是事后编写的，用以理解一个系统：也许是为了寻找错误，也许是因为需要修改该系统。

一个形式化规约就像一张详细的图纸；非形式化规约则像一张设计草图。**FIXME:** 对于像为一所房子加一个天窗或者一扇门这样的小型工程，使用设计草图可能就够了；对于简单系统模型来说，使用非形式化规约可能也足够了。编写形式化规约的最大好处在于可以使用工具检查它是否有错。本书教你如何编写并且检查形式化规约。学习编写形式化规约对编写非形式化规约也是有帮助的。

## 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

?

←

→

C

I

S

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<sup>+</sup>.

The TLA<sup>+</sup> 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<sup>+</sup> 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<sup>+</sup> specification that can be checked with the TLA<sup>+</sup> tools.

PlusCal is more convenient than TLA<sup>+</sup> 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<sup>+</sup> makes it better for these algorithms. However, TLA<sup>+</sup> 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<sup>+</sup> specification. PlusCal was designed so the TLA<sup>+</sup> 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<sup>+</sup> 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<sup>+</sup> 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<sup>+</sup> 规约。在 TLA<sup>+</sup> 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**。

将初始谓词 `Init1` 与次态关系 `Next1` 填入相应的文本域，并[运行 TLC](#)。TLC 将在几秒钟后停下来，并且没有发现错误。这意味着该规约是合理的，更准确地说，它完全确定了一组行为。

让我们修改规约，以使得它不能确定一组行为。切换到模块编辑器（可点击相应标签页），将 `Next1` 定义中的 `∧ b' = 0` 修改为 `∧ b' = "xyz"`。现在，`Next1` 的第二个析取式允许**FIXME: 一个步骤将 b 从数字 0 设置为字符串**“xyz”。保存模块，回到模型编辑器，再次运行 TLC。这次，TLC 将报告如下错误：

Attempted to check equality of string "xyz" with non-string: 0  
(试图检查字符串"xyz" 与非字符串 0 的相等性)

TLC Errors 窗口如下所示：

你可以 调整 TLC Errors 视图中文本域的大小。

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

它描述了一条**FIXME: 错误踪迹**：

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

这是 TLC 在遇到错误之前**FIXME: 所构造的行为的开头部分**。变量 b 的值“xyz”带有淡红色背景，这表示该值与前一个状态中 b 的取值不同。双击错误踪迹中的如下行，

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

将**FIXME: 跳转到模块编辑器的如下部分**：

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

高亮显示的部分是次态动作 Next1 中允许  $b = 1 \rightarrow b = \text{"xyz"}$  步骤的析取式。

要计算满足  $b = \text{"xyz"}$  的状态 **FIXME: with** 的所有可能的后继状态，TLC 需要计算公式  $\text{"xyz"} = 0$  的值。（错误信息的剩余部分表明 TLC 正在计算该公式，以便决定定义 Next1 中的子公式  $b = 0$  的真假。）TLC 无法完成该工作，因为  $\text{TLA}^+$  的语义没有指明一个字符串是否等于一个数字。所以，TLC 无法确定公式  $\text{"xyz"} = 0$  的值是 TRUE 还是 FALSE，因此它报告了一个错误。

**FIXME: Why shouldn't "xyz" be unequal to 0?**

将“xyz”替换为 0 以恢复 Next1 的初始定义，然后保存模块。回到模型编辑器，运行 TLC。TLC 将再次报告没有错误。

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  $\text{TLA}^+$  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:

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

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

(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  $\text{TLA}^+$  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

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

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

`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.

C

I

S

## 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

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

?

We substitute 1 for b and simplify as follows.

$$\vee \wedge 1 = 0 \quad \text{the formula obtained by substituting 1 for b in Next1.}$$

$$\wedge b' = 1$$

$$\vee \wedge 1 = 1$$

$$\wedge b' = 0$$

$$= \vee \wedge \text{FALSE} \quad \text{because } (0 = 1) = \text{FALSE} \text{ and } (1 = 1) = \text{TRUE}$$

$$\wedge b' = 1$$

$$\vee \wedge \text{TRUE}$$

$$\wedge b' = 0$$

$$= \vee \text{FALSE} \quad \text{because } \text{FALSE} \wedge F = \text{FALSE} \text{ and } \text{TRUE} \wedge F = F$$

$$\vee b' = 0 \quad \text{for any truth value F}$$

$$= b' = 0 \quad \text{because } \text{FALSE} \vee F = F \text{ for any truth value F.}$$

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**

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

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:

?

←

→

C

I

S

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  $b$ .

**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.

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

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  $b$ :

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

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.

Why is this formula true if  $b$  equals 42?

?

←

→

C

I

S



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' = 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.

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

?

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 {
}
```

→

C

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.

I

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

S

```
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<sup>+</sup> 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 ≜ b ∈ {0, 1}
Next ≜ IF b = 0 THEN b' = 1
      ELSE 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 `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

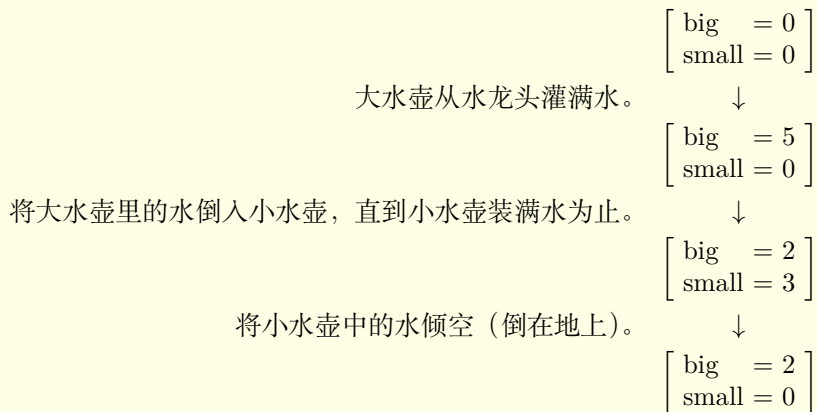
S

## 3 Die Hard 问题

在电影《虎胆龙威（三）》中，两位英雄必须解决这样一个问题：用一个 5 加仑的水壶，一个 3 加仑的水壶，**FIXME: 和一个水龙头**，如何得到 4 加仑的水？现在，我们用  $TLA^+$  和 TLC 模型验证器解决该问题。

### 3.1 用 TLAPlus 描述 Die Hard 问题

解决该问题的首要任务是以数学的方式将由英雄、水壶和水龙头构成的物理系统建模为一个离散系统。在该物理系统中，唯一相关的状态是两个水壶中的水量。因此，我们用两个变量建模该系统：big 和 small 分别表示大小两个水壶中的水量。选定变量之后，弄清如何编写规约的一个好方法是先写系统的某个行为的头几个状态。一开始，水壶是空的，因此 big 和 small 都等于 0。系统的某个行为的开头可能如下所示（一个状态是一种变量赋值。在本例中，变量是 big 和 small。）：



稍作思考，我们会发现一个行为包含三类步骤：

- （从水龙头）装满某个水壶。
- 倾空某个水壶。
- 从一个水壶往另一个水壶倒水。这分两种情况：
  - 第一个水壶被倾空。
  - 第二个水壶被装满。此时，第一个水壶中可能还有水。

现在，我们可以编写规约了。在工具箱中[打开一个新的规约](#)，命名为 DieHard。由于该规约需要用到算术操作，所以它以

开头。我们声明变量，编写初始谓词，**FIXME: 并按照惯例**将初始谓词命名为 Init。

VARIABLES big, small

Init  $\triangleq$   $\bigwedge \text{big} = 0$   
 $\bigwedge \text{small} = 0$

VARIABLES  $\sqcup \text{big}, \sqcup \text{small}$

Init $\sqcup = \sqcup / \sqcup \text{big} \sqcup = \sqcup 0$   
 $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \text{small} \sqcup = \sqcup 0$

?

←

→

C

I

S

每一类可能的步骤都有两种情况，分别对应于不同的水壶（对于第三类步骤，第一个水壶可以是两个水壶中的任意一个）。这提示我们将次态动作写成 6 个公式的析取式，每个公式对应一类可能的步骤。因此，我们如下定义次态动作，并按照惯例将其命名为 Next：

Next  $\triangleq$   $\bigvee \text{FillSmall}$   
 $\bigvee \text{FillBig}$   
 $\bigvee \text{EmptySmall}$   
 $\bigvee \text{EmptyBig}$   
 $\bigvee \text{SmallToBig}$   
 $\bigvee \text{BigToSmall}$

$\sqcup \sqcup \text{Next} \sqcup = \sqcup \sqcup / \sqcup \text{FillSmall}$   
 $\sqcup \sqcup \sqcup / \sqcup \text{FillBig}$   
 $\sqcup \sqcup \sqcup / \sqcup \text{EmptySmall}$   
 $\sqcup \sqcup \sqcup / \sqcup \text{EmptyBig}$   
 $\sqcup \sqcup \sqcup / \sqcup \text{SmallToBig}$   
 $\sqcup \sqcup \sqcup / \sqcup \text{BigToSmall}$   
 $\sqcup \sqcup$

这 6 个公式——通常称为次态动作的子动作——的定义必须出现在 Next 的定义之前。（在 TLA<sup>+</sup>中，一个符号需要先定义或者先声明之后才可使用。）现在，我们定义这些子动作。

大多数程序员期望看到如下定义的 FillSmall 公式：

FillSmall  $\triangleq$   $\text{small}' = 3$

该公式当然可以被如下所示的步骤所满足：

$\left[ \begin{array}{l} \text{big} = 2 \\ \text{small} = 1 \end{array} \right] \rightarrow \left[ \begin{array}{l} \text{big} = 2 \\ \text{small} = 3 \end{array} \right]$

但是，它也可以被步骤

$\left[ \begin{array}{l} \text{big} = 2 \\ \text{small} = 1 \end{array} \right] \rightarrow \left[ \begin{array}{l} \text{big} = \sqrt{42} \\ \text{small} = 3 \end{array} \right]$

所满足，这是因为在公式中施行替换

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

会产生永真式  $3 = 3$ 。由于“将小水壶装满”这一步骤应该保持大水壶里的水量不变，所以子动作 FillSmall 必须保证  $\text{big}'$  等于  $\text{big}$ 。**FIXME: 基于这种观察**，前四个子动作的定义是显而易见的：

?  
←  
→  
C  
I  
S

FillSmall	$\triangleq \wedge \text{small}' = 3$ $\wedge \text{big}' = \text{big}$	$\llbracket \text{FillSmall} \rrbracket = \llbracket \wedge \text{small}' = 3$ $\wedge \text{big}' = \text{big} \rrbracket$
FillBig	$\triangleq \wedge \text{big}' = 5$ $\wedge \text{small}' = \text{small}$	$\llbracket \text{FillBig} \rrbracket = \llbracket \wedge \text{big}' = 5$ $\wedge \text{small}' = \text{small} \rrbracket$
EmptySmall	$\triangleq \wedge \text{small}' = 0$ $\wedge \text{big}' = \text{big}$	$\llbracket \text{EmptySmall} \rrbracket = \llbracket \wedge \text{small}' = 0$ $\wedge \text{big}' = \text{big} \rrbracket$
EmptyBig	$\triangleq \wedge \text{big}' = 0$ $\wedge \text{small}' = \text{small}$	$\llbracket \text{EmptyBig} \rrbracket = \llbracket \wedge \text{big}' = 0$ $\wedge \text{small}' = \text{small} \rrbracket$
		$\llbracket \text{C} \rrbracket$

后两个子动作 SmallToBig 和 BigToSmall 的定义稍显复杂，因为每个都有两种情况。考虑 SmallToBig。我们可以将它的两种情况表示成两个公式的析取：

$$\begin{aligned} \text{SmallToBig} \triangleq & \vee \wedge \text{big} + \text{small} > 5 \\ & \wedge \text{big}' = 5 \\ & \wedge \text{small}' = \text{small} - (5 - \text{big}) \\ & \vee \wedge \text{big} + \text{small} \leq 5 \\ & \wedge \text{big}' = \text{big} + \text{small} \\ & \wedge \text{small}' = 0 \end{aligned}$$

如果大水壶装不下小水壶中所有的水，那么从小水壶中倒入大水壶中的水量是  $5 - \text{big}$  加仑。

这个定义还不错，但是可以表达得更紧凑些。注意到 SmallToBig 步骤 将 big 设置为  $\text{big} + \text{small}$  与 5 中的较小值。我们定义 Min，使得  $\text{Min}(m, n)$  等于  $m$  与  $n$  的较小值（如果  $m$  与  $n$  是数字）。

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

因为从小水壶中倒出的水量等于大水壶增加的水量，所以我们可以如下定义 SmallToBig：

$$\begin{aligned} \text{SmallToBig} \triangleq & \wedge \text{big}' = \text{Min}(\text{big} + \text{small}, 5) \\ & \wedge \text{small}' = \text{small} - (\text{big}' - \text{big}) \end{aligned}$$

这个定义有一个缺陷。当阅读一个动作公式时，我们通常想知道一个特定变量的值是如何变化的。如果带撇变量的值被表示成无撇变量的值的函数，那么这 **FIXME: 很容易** 做到。但是，该定义用  $\text{big}'$ （以及  $\text{big}$  与  $\text{small}$ ）的值来表示  $\text{small}'$  的值。我们可以通过重写定义修复这个缺陷：

$$\begin{aligned} \text{SmallToBig} \triangleq & \wedge \text{big}' = \text{Min}(\text{big} + \text{small}, 5) \\ & \wedge \text{small}' = \text{small} - (\text{Min}(\text{big} + \text{small}, 5) - \text{big}) \end{aligned}$$

不过，最好不要重复使用表达式  $\text{Min}(\text{big} + \text{small}, 5)$ 。我发现使用 **FIXME: 从一个水壶到另一个水壶的倒水量** 来表达该动作更为简练。我更喜欢使用  $\text{TLA}^+$  提供的 **LET / IN 结构** 将它写成如下形式（LET / IN 允许我们 **FIXME: 在表达式内部给出局部定义**）：

```

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

```

```

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

```

(注意 poured 等于  $\text{Min}(\text{small}, 5 - \text{big})$ 。) 子动作 BigToSmall 的定义与之类似。

```

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

```

```

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

```

我们还应该定义一个类型不变式。显然，变量 big 和 small 的值都是自然数，并且  $\text{big} \leq 5$ ,  $\text{small} \leq 3$ 。为了 **FIXME: 表达这一点**，我们使用在 Integers 模块中定义的  $\dots$  操作符。 $i..j$  表示从  $i$  到  $j$  的整数集合。更精确地讲， $i..j$  被定义为满足  $i \leq k$  与  $k \leq j$  条件的整数  $k$  构成的集合。如果  $j < i$ ，则  $i..j$  是空集。操作符  $\dots$  的定义是：

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

$\{x \in S : P(x)\}$  是  $S$  中所有满足  $P(x)$  的元素  $x$  构成的  $S$  的子集。□

这里，Int 是所有整数构成的集合，定义在 Integers 模块中。类型不变式的定义如下所示：

```

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

```

```

TypeOK ==  $\wedge$  big  $\in 0..5$ 
           $\wedge$  small  $\in 0..3$ 

```

该定义最好紧跟在变量 big 与 small 的声明语句之后。

## 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<sup>+</sup> specifications a little more fun. But could a TLA<sup>+</sup> 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  $\text{big} \neq 4$  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.

How to type  $\neq$ .  $\square$

?

←

→

C

I

S

### 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<sup>+</sup> 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

```

if ( big + small > 5 ) { small := small - (5 - big);
                        big := 5
}
else { big := big + small;
      small := 0
}

```

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  $\text{TLA}^+$  and write:

```

big    := big + poured;
small := small - poured

```

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

```

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

```

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

In the  $\text{TLA}^+$  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:

```

either big := 5
or      small := 3
...
or      with ( poured = Min(big + small, 3) - small )
        { big    := 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  $\text{TLA}^+$  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 算法

Euclid 算法是计算两个正整数的最大公约数（简称 *gcd*）的经典算法。我们**FIXME**: 考虑 Euclid 在《几何原本》中描述的算法的一个简单低效的版本。不过，在设计算法计算最大公约数之前，我们需要先精确定义什么是最大公约数。

如果你不熟悉量词  $\forall$  和  $\exists$ ，请看[这里](#)。□

### 4.1 The Greatest Common Divisor

We want to define an operator  $\text{GCD}$  such that  $\text{GCD}(m, n)$  equals the  $\text{gcd}$  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  $\text{gcd}$  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  $\text{GCD}$  and any other related definitions and properties we might need.

Why we usually don't re-use specifications in practice.

In the Toolbox, open a new specification called  $\text{GCD}$ . (TLA<sup>+</sup> 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<sup>+</sup> 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  $\text{Divides}$  so that  $\text{Divides}(p, n)$  equals  $\text{TRUE}$  if the integer  $p$  divides the integer  $n$ , and equals  $\text{FALSE}$  if it doesn't. You learned in grade school that  $p$  divides  $n$  iff  $n/p$  is an integer. The  $\text{Integers}$  module defines  $\text{Int}$  to be the set of all integers. So, an obvious definition of  $\text{Divides}$  is

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

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

The  $\text{Integers}$  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  $\text{Reals}$  module that defines ordinary division, but

?

←

→

C

I

S

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

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

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  $\text{Divides}(2, 4)$ . This produces an error message that looks like:

[How to use TLC to evaluate a constant expression.](#)

```
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  $\exists x \in S : \text{exp}$  by computing all the elements in the set  $S$  and evaluating  $\text{exp}$  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

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

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  $\text{Divides}(2, 4)$  obtains the value `TRUE`. Check that TLC calculates  $\text{Divides}(2, 5)$  to equal `FALSE`.

[How to override a definition in TLC.](#)

The  $\text{gcd}$  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  $\text{DivisorsOf}(n)$  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\$](#)

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

Add this definition to module `GCD` and have TLC evaluate  $\text{DivisorsOf}(493)$ . 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  $\text{TLA}^+$  `CHOOSE` operator. The expression

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

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

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

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

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

since there is no integer whose square equals  $-4$ .

[Learn more about CHOOSE here.](#)  $\square$

Using  $\text{CHOOSE}$ , 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  $S$ :

$$\begin{aligned} \text{SetMax}(S) &\triangleq \\ &\text{CHOOSE } i \in S : \forall j \in S : i \geq j \end{aligned} \qquad \begin{aligned} \text{SetMax}(S) &\triangleq \bigcup \\ &\bigcup \bigcup \text{CHOOSE } i \in \text{in } S : \bigcup \bigcup A \cup j \in \text{in } S : \bigcup i \geq j \end{aligned}$$

Note that  $\geq$  is typed  $\geq$ . It can also be typed  $\text{\textbackslash geq}$ . Add this definition to module  $\text{GCD}$  and check it by having the Toolbox evaluate the expression  $\text{SetMax}(\text{DivisorsOf}(493))$ , 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  \$\cap\$ , detour here.](#)  $\square$

$$\begin{aligned} \text{GCD}(m, n) &\triangleq \\ &\text{SetMax}(\text{DivisorsOf}(m) \cap \text{DivisorsOf}(n)) \end{aligned} \qquad \begin{aligned} \text{GCD}(m, n) &\triangleq \bigcup \\ &\bigcup \bigcup \text{SetMax}(\text{DivisorsOf}(m) \cap \text{DivisorsOf}(n)) \end{aligned}$$

Add this definition to module  $\text{GCD}$  and check that it's correct by evaluating  $\text{GCD}$  for some numbers. You will find that  $\text{TLC}$  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  $\text{GCD}(m, n)$  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.

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

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`.

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

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

There are two ways to write comments in  $\text{TLA}^+$ . 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.

Give it a try.

```
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.

What does `Divides(p,n)` mean if `p` and `n` are not integers—or not even numbers?

### 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](#) `\` as `Nat \ {0}`. Our assumption about M and N can therefore be written as follows:

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

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

**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.

?

←

→

C

I

S

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

$$\text{variables } x = M, y = N;$$

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 ( $x \neq y$ ) { if ( $x < y$ ) { else     {x := x - y} } }	$\sqcup\{\sqcup\texttt{while}_{\sqcup}(x_{\sqcup}\#y_{\sqcup})\sqcup\{\sqcup\texttt{if}_{\sqcup}(x_{\sqcup}<y_{\sqcup})\sqcup\{\sqcup y_{\sqcup}:=\sqcup y_{\sqcup}-\sqcup x_{\sqcup}\}\sqcup\{\sqcup\texttt{else}_{\sqcup}\sqcup\{\sqcup x_{\sqcup}:=\sqcup x_{\sqcup}-\sqcup y_{\sqcup}\}\sqcup\{\sqcup\}\}$
--	--

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<sup>+</sup> specification. Do this with the File menu's **Translate PlusCal Algorithm** command, or by typing **control+t**. The translator inserts the TLA<sup>+</sup> 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 TLAPlus Translation

The TLA<sup>+</sup> 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  $\text{TLA}^+$ , 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.

$$\text{vars} \triangleq \langle x, y, pc \rangle$$

In TLA<sup>+</sup>, tuples are enclosed between angle brackets `<` and `>`, which are typed `Tuples are explained here.` `<<` and `>>`, so the definition of `vars` is written

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

Next comes the definition of the initial predicate.

$$\begin{aligned} \text{Init} &\triangleq \wedge x = M \\ &\quad \wedge y = N \\ &\quad \wedge pc = \text{"Lbl\_1"} \end{aligned}$$

I have reformatted the translation slightly to make it a bit easier to read.

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.

Here is a pop-up window with this definition.

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<sup>+</sup> 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

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

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 “LbL1”. As long as  $x \neq y$ , it can execute LbL1 steps that leave pc equal to “LbL1” and decrease  $x$  or  $y$ . If  $x = y$ , it can execute an LbL1 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 LbL1 to be the algorithm’s next-state action. However, the translation defines Next to be the disjunction of LbL1 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  $\text{Init} \wedge \square[\text{Next}]_{\text{vars}}$ , where vars is defined to be the triple  $\langle x, y, \text{pc} \rangle$  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  $\text{TLA}^+$  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  $\text{GCD}(M, N)$ .
- 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.

[What are safety and liveness properties?](#)

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):

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

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

?

←

→

C

I

S

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:

<div style="background-color: #e0e0ff; padding: 2px; margin-bottom: 2px;">?</div> <div style="background-color: #e0e0ff; padding: 2px; margin-bottom: 2px;">←</div> <div style="background-color: #e0e0ff; padding: 2px; margin-bottom: 2px;">→</div> <div style="background-color: #e0e0ff; padding: 2px; margin-bottom: 2px;">C</div> <div style="background-color: #e0e0ff; padding: 2px; margin-bottom: 2px;">I</div> <div style="background-color: #e0e0ff; padding: 2px; margin-bottom: 2px;">S</div>	$\text{PartialCorrectness} \triangleq$ $(pc = \text{"Done"}) \Rightarrow (x = y) \wedge (x = \text{GCD}(M, N))$	$\text{PartialCorrectness}_{\sqcup} \triangleq$ $\sqcup (pc_{\sqcup} = \text{"Done"})_{\sqcup} \Rightarrow \sqcup (x_{\sqcup} = y)_{\sqcup} / \wedge \sqcup (x_{\sqcup} = \text{GCD}(M, \sqcup N))$
---	---	---

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).

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

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

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

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:

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

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  $\text{GCD}(x0, y0)$  when evaluating the invariant. Doing that requires it to compute  $\text{Divisors}(x0)$  and  $\text{Divisors}(y0)$ . TLC computes  $\text{Divisors}(n)$  from the definition of  $\text{Divisors}$  by enumerating all the elements  $p$  in  $\text{Int}$  and checking if  $\text{Divides}(p, n)$  is true. In the common case when  $p$  does not divide  $n$ , this computing  $\text{Divides}(p, n)$  requires TLC to check that  $n$  does not equal  $p * q$  for every element  $q$  of  $\text{Int}$ . Since our model redefines  $\text{Int}$  to be a set with about 2000 elements, computing  $\text{GCD}(x0, y0)$  requires TLC to compute an expression of the form  $n = p * q$  about 8 million times. It computes  $\text{GCD}(x0, y0)$  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  $1 \dots n$ . TLC will therefore correctly compute  $\text{GCD}(x0, y0)$  for  $x0$  and  $y0$  in  $1 \dots N$  if we redefine  $\text{Int}$  to equal  $1 \dots N$ . Change the model to override the definition of  $\text{Int}$  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)  $\wedge$  (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:

If you got a different error, [click here](#).

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 disappears, 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  $s_1 \rightarrow s_2 \rightarrow \dots$  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  $s_i \rightarrow s_{i+1}$  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  $s_1 \rightarrow \dots \rightarrow s_{n+1}$  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  $\text{WF}_{\text{vars}}(\text{Next})$  (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

$$\text{Next} \triangleq \text{LbL1} \vee (\text{pc} = \text{"Done"} \wedge \text{UNCHANGED vars})$$

where `vars` is defined to equal  $\langle x, y, x0, y0, \text{pc} \rangle$ . Action `LbL1` 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.

?

←

→

C

I

S

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 PlusCal 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.

?

←

→

C

I

S

## 4.8 The Grain of Atomicity

The TLA<sup>+</sup> 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 `Lbl1` 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 `Lbl1` has become formula `abc`, whose definition is the same as the original definition of `Lbl1` except that the string “`Lbl1`” 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: while (  $x \neq y$  ) { d: if (  $x < y$  ) {  $y := y - x$  }
                                else      {  $x := x - y$  }
                                } ;
assert (  $x = y$  )  $\wedge$  (  $x = \text{GCD}(x_0, y_0)$  )

```

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<sup>+</sup> 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<sup>+</sup> Proof Track](#)<sup>□</sup>.

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.

?

←

→

C

### 4.9.1 Proving Invariance

I

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

S

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

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  $s_1 \rightarrow s_2 \rightarrow \dots$ , we prove:

1. Inv is true in state  $s_1$ .
2. For every step  $s_n \rightarrow s_{n+1}$  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:

$$I1. \text{Init} \Rightarrow \text{Inv}$$

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

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

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

$$I2. \text{Inv} \wedge \text{Next} \Rightarrow \text{Inv}'$$

?

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

←

→

C

I

S

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  $42 = \text{GCD}(M, N)$  (because  $pc' = \text{"Done"}$  and  $x' = y'$  both equal TRUE). Hence with these substitutions, I2 becomes  $\text{TRUE} \wedge \text{TRUE} \Rightarrow (42 = \text{GCD}(M, N))$ , which equals  $42 = \text{GCD}(M, N)$ . 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<sup>+</sup> 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.  $\text{Inv} \Rightarrow \text{PartialCorrectness}$

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:

$$\text{GCD}(x, y) = \text{GCD}(M, N)$$

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:

$$\begin{aligned} \text{Inv} \triangleq & \quad \wedge \text{GCD}(x, y) = \text{GCD}(M, N) \\ & \quad \wedge (\text{pc} = \text{“Done”}) \Rightarrow (x = y) \end{aligned}$$

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:

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

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

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

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

[Click here](#) 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).

?

←

→

C

I

S

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:

$$\begin{aligned} \text{TypeOK} &\triangleq \bigwedge x \in \text{Nat} \setminus \{0\} \\ &\quad \bigwedge y \in \text{Nat} \setminus \{0\} \end{aligned}$$

and changing the definition of `Inv` to

$$\begin{aligned} \text{Inv} &\triangleq \bigwedge \text{TypeOK} \\ &\quad \bigwedge \text{GCD}(x, y) = \text{GCD}(M, N) \\ &\quad \bigwedge (\text{pc} = \text{“Done”}) \Rightarrow (x = y) \end{aligned}$$

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 the **if** statement `d`. Show that the formula `Inv` defined above is not an inductive ANSWER

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](#)

$\langle \text{GCD1}, \text{GCD2}, \text{GCD3} \rangle$   $\langle \langle \text{GCD1}, \text{GCD2}, \text{GCD3} \rangle \rangle$

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

$$\forall \dots \in \text{Nat} \setminus \{0\} : \dots$$

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..5—and run TLC on it. If you’ve made no error, it should report the value  $\langle \text{TRUE}, \text{TRUE}, \text{TRUE} \rangle$ . 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..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<sup>+</sup> versions of the invariance proof of Euclid’s algorithm and the proof of GCD1–GCD3 are in Section 11 of the [Proof Track](#)<sup>□</sup>.



### 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  $s \rightarrow t$  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  $\text{Next}$  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:

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

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

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  $\text{Inv}$ .

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  $\text{SetGCD}(T)$  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  $T$ :

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

[ASCII version](#)

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  $\text{Cardinality}(S) > 1$ . 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  $\triangleq$   $\wedge$  pc = "Lbl_1"
 $\wedge$  IF Cardinality(S) > 1
  THEN  $\wedge \exists x \in S :$ 
     $\exists y \in \{s \in S : s > x\} :$ 
       $S' = ((S \setminus \{y\}) \cup \{y - x\})$ 
     $\wedge$  pc' = "Lbl_1"
  ELSE  $\wedge$  pc' = "Done"
     $\wedge S' = S$ 

```

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

**with** (  $v_1 \in S_1, \dots, v_i \in S_i$  ) {  $\Sigma$  }

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

is executed by letting each  $v_i$  be arbitrary element in  $S_i$  and executing  $\Sigma$  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:

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

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

$$\text{SetGCD}(S) = \text{SetGCD}(\text{Input})$$

A rigorous proof uses the inductive invariant

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

where the type invariant is defined by

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

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:

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

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

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

ASCII version

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<sup>+</sup>, 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(\_)

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:

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

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 [ANSWER](#) following important result from number theory. The gcd of a set  $\{n_1, \dots, n_k\}$  of positive integers equals  $i_1 * n_1 + \dots + i_k * n_k$  for some integers  $i_j$ .

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

?

←

→

C

I

S