The *Principles* Track

7 Mutual Exclusion

7.1 The Problem

The mutual exclusion problem was introduced by Edsger Dijkstra in his article *Solution of a Problem in Concurrent Control*, published in the *Communications of the ACM*, Volume 8, Number 9 (September, 1965), page 569. This seminal article launched the field of concurrent algorithms. Here is how Dijkstra began his statement of the problem.

> [C]onsider N computers, each engaged in a process which, for our aims, can be regarded as cyclic. In each of the cycles a so-called "critical section" occurs and the computers have to be programmed in such a way that at any moment only one of these N cyclic processes is in its critical section.

Dijkstra wrote about multiple computers, but the application he had in mind was multiple processes running on the same computer. The mutual exclusion problem has come to be stated in terms of processes rather than computers, so that is the terminology that we will use. The property that there is never more than one process in its critical section is called *mutual exclusion*. It is, of course, an invariance property.

What is a process?

Dijkstra next stated what operations the processes could use.

> In order to effectuate this mutual exclusion of critical-section execution the computers can communicate with each other via a common store. Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order.

Dijkstra disliked the anthropomorphic term *memory* and usually wrote *store* instead.

Mutual exclusion is central to modern concurrent programming. Today, multiprocess computers provide special instructions for implementing mutual exclusion. That was not the case in 1965. The only interprocess communication primitives were reading and writing a shared memory register—operations that Dijkstra assumed to be atomic actions. Today, the *mutual exclusion problem* is not restricted to solutions based only on reading and writing shared memory. For example, there are distributed solutions in which processes communicate with messages. Of course, the problem becomes trivial if we can use sufficiently powerful communication primitives.

Dijkstra next stated four requirements that a solution must satisfy. The first was:

(a) The solution must be symmetrical between the N computers; as a result we are not allowed to introduce a static priority.

Requirement (a) is the only part of this article that has turned out not to be important, and it has been ignored. We too can ignore it. The next requirement was:

(b) Nothing may be assumed about the relative speeds of the N computers; we may not even assume their speeds to be constant in time.

This requirement is now taken for granted when studying concurrent algorithms, and it requires no discussion. Explicitly stating it for the first time is one of the major contributions of the article. The next requirement was:

(c) If any of the computers is stopped well outside its critical section, this is not allowed to lead to potential blocking of the others.

The part of a process's code "well outside its critical section" is now called the *noncritical section*. Implicit in this requirement is that a process is allowed to stop in its noncritical section. Requirement (c) is a fundamental part of the mutual exclusion problem. Removing it makes the problem much easier, both in principle and in practice. For example, it is quite easy to write an algorithm in which processes take turns entering the critical section. This satisfies mutual exclusion, but stopping one process prevents the other from entering its critical more than one additional time, so it doesn't satisfy (c). In fact, we have already written such an algorithm.

Problem 7.1 Show that a solution to the alternation problem, as described in Section 6.9□, satisfies the mutual exclusion condition for $N = 2$, where the *put* and *get* operations are the critical sections. Generalize this to show that that a round-robin synchronization algorithm (Section 6.10□) satisfies mutual exclusion for an arbitrary positive integer N. HINT

Dijkstra's final requirement was:

(d) If more than one computer is about to enter its critical section, it must be impossible to devise for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity. In other words, constructions in which "After you"–"After you"—blocking is still possible, although improbable, are not to be regarded as valid solutions.

This asserts a liveness property that the algorithm must satisfy. Today, this property would be expressed as: if at least one process is trying to enter its critical section, then some process must eventually enter its critical section. However, it would be 10 years before the concepts of safety and liveness were identified, and the meaning of "eventually" would not have been clear to readers at the time. Dijkstra explained the requirement in a somewhat roundabout way by saying what was not allowed, an explanation that readers were sure to understand.

What Dijkstra called *after-you, after-you blocking* is now known as *livelock*. Requirement (d) is now called *deadlock freedom*. This is a somewhat confusing name, since deadlock usually means a state in which no process can take a step, which is not the case in the *livelock* that Dijkstra ruled out.

Deadlock freedom assures only that some process enters its critical section. It doesn't assure that any particular process does. It allows the possibility that some process waits forever trying to enter its critical section while other processes keep entering and leaving their critical sections. Such a process is said to be *starved*. (The metaphor of starvation comes from the *dining philosophers problem*, another multiprocess synchronization problem invented by Dijkstra.) A stronger requirement than deadlock freedom is *starvation freedom*, which requires that any process that tries to enter its critical section eventually does so. The solution that Dijkstra presented in his 1965 paper is deadlock free but not starvation free.

7.2 The One-Bit Protocol

7.2.1 The Protocol

Dijkstra's original algorithm uses a simple protocol for ensuring mutual exclusion that appears in a number of subsequent algorithms. I call it the *One-Bit Protocol*. Here is how it works in the case of two processes: Each process maintains a Boolean value. It can enter its critical section only by setting its value to TRUE and reading the other process's variable equal to FALSE.

Let's write the One-Bit Protocol more precisely. Let the processes be named 0 and 1 and let their Boolean values be $x[0]$ and $x[1]$. Using PlusCal notation, the algorithm of process *self* is as follows. (Note that $1 - self$ is the process other than process *self*.)

$e1$: $x[self]$:= TRUE ;
$e2$: **if** ($\neg x[1-self]$) { cs: critical section }

We don't specify the critical section, except that we assume it does not change the value of the array (function) x. In particular, it need not be an atomic action. (In PlusCal notation, it could contain labels.)

It's easy to see that this protocol ensures mutual exclusion. Here's a simple proof by contradiction.

> Assume that both processes are in their critical sections. The first one that entered its critical section, call it process i, did so after setting $x[i]$ to TRUE. Process $1 - i$ entered the critical section after i, so it read $x[i]$ in its **if** test after process i had set $x[i]$ to TRUE. Hence the read of $x[i]$ by process $1 - i$ in its **if** test obtained the value TRUE, so it couldn't have entered its critical section—contradicting the assumption that both processes are in their critical section.

This is a correct and convincing proof. It is a *behavioral* proof, based on reasoning about the order in which operations are executed. This proof is quite informal. Behavioral proofs can be made more formal, but I don't know any practical way to make them completely formal—that is, to write executable descriptions of real algorithms and formal behavioral proofs that they satisfy correctness properties. This is one reason why, in more than 35 years of writing concurrent algorithms, I have found behavioral reasoning to be unreliable for more complicated algorithms. I believe another reason to be that behavioral proofs are inherently more complex than state-based ones for sufficiently complex algorithms. This leads people to write less rigorous behavioral proofs for those algorithms—especially with no completely formal proofs to serve as guideposts.

To avoid mistakes, we have to think in terms of states, not in terms of executions. So, I will show you how to write a state-based proof. Since I have not precisely specified the One-Bit Protocol (for example, by writing a complete PlusCal algorithm), no proof of its correctness can be completely formal. However, even the informal state-based proof is long and boring. Please don't be discouraged by this. It's important to learn how to write a very rigorous state-based proof, because that's the kind of proof you'll have to write if you want to be sure that a more complicated algorithm is correct. Exciting proofs concentrate on the interesting parts that display insight, skimming over unimportant details. Unfortunately, our insight is often not as good as we think, and we can too easily miss a fatal flaw lurking in neglected details. Although finding a state-based proof requires insight, checking it does not. A careful attention to details is all it takes to avoid mistakes.

Still, behavioral reasoning provides a different way of thinking about an algorithm, and thinking is always helpful. Behavioral reasoning is bad only if it is used instead of state-based reasoning rather than in addition to it.

7.2.2 An Assertional Proof

Mutual exclusion is an invariance property. Let $InCS(i)$ be the state predicate that is true iff process i is in its critical section. Mutual exclusion asserts the invariance of the state predicate *MutualExclusion*, defined for our two processes by

$$MutualExclusion \triangleq \neg(InCS(0) \land InCS(1))$$

Suppose that the complete algorithm is described by an initial predicate *Init* and next-state action *Next*. We saw in Section 4.9.1 that we prove a formula *Inv* to be an invariant of the algorithm by proving:

I1. $Init \Rightarrow Inv$

I2. $Inv \land Next \Rightarrow Inv'$

Condition I1 asserts that *Inv* is true initially, and condition I2 asserts that any single step of the algorithm executed when *Inv* is true leaves *Inv* true. A state predicate *Inv* that satisfies condition I2 is called an *inductive invariant* of the algorithm. (More precisely, it's an inductive invariant of the next-state action *Next*.)

This is not quite correct.

To prove that *MutualExclusion* is an invariant of the algorithm, we find an invariant *Inv* satisfying 1 and 2 such that the following condition also holds:

I3. $Inv \Rightarrow MutualExclusion$

If *Inv* is an invariant, meaning it is true of every state of every behavior of the algorithm, then condition I3 implies that *MutualExclusion* is also an invariant. Thus, to prove that *MutualExclusion* is an invariant, we have to find an inductive invariant *Inv* that is true initially and implies *MutualExclusion*. Before reading further, see if you can find such an invariant *Inv*.

We don't expect to be able to deduce anything about what a step might do in a state not satisfying a type invariant, so *Inv* will have a type invariant *TypeOK* as a conjunct. Since, we want *Inv* to imply *MutualExclusion*, our first guess might be to let *Inv* equal $TypeOK \land MutualExclusion$. However, this is not an inductive invariant. Consider a state in which process 0 is at $e2$, process 1 is in its critical section, and $x[1]$ equals FALSE. This state satisfies $TypeOK \land MutualExclusion$, but an $e2$ step by process 0 in that state makes *MutualExclusion* false.

The invariance of *MutualExclusion* depends on $x[i]$ equaling TRUE when process i is in its critical section. So, our next attempt is to let *Inv* equal

$$TypeOK \ \land \ MutualExclusion \ \land \ \forall i \in \{0,1\} : InCS(i) \Rightarrow x[i]$$

However, this formula is still not an inductive invariant. Consider a state satisfying this formula in which neither process is in its critical section, $x[0]$ and $x[1]$ equal FALSE, and some process i is at $e2$. That process can then take an $e2$ step making $InCS(i)$ true and leaving $x[i]$ false, making the formula false. An inductive invariant must also assert that $x[i]$ is TRUE if process i is at $e2$. Remember that, for a PlusCal algorithm, process i is at label $e2$ iff $pc[i]$ equals "e2". Modifying the last conjunct to assert this, we get:

$$\begin{array}{l} \land \ TypeOK \\ \land \ MutualExclusion \\ \land \ \forall \, i \in \{0,1\} \ : \ InCS(i) \lor (pc[i] = \text{“e2”}) \Rightarrow x[i] \end{array}$$

Further thought reveals no states satisfying this predicate from which a step of the protocol can make the predicate false. This doesn't mean that there are no such states; it just means that we can't think of them. The only way to be sure that there are none is by proving that this is the required inductive invariant. So,

we define *Inv* to equal this formula and try to prove that it satisfies conditions I1–I3. It turns out that it does; but if it didn't, the proof would reveal how it needed to be changed.

Before we write the proof, observe how *Inv* was constructed. We started with the type invariant and the invariant *MutualExclusion* that we want to prove. We then kept strengthening the invariant when we found a state satisfying the formula that permitted a step that makes the formula false. This is a standard method for finding an inductive invariant. With experience, you will learn to see right away most of the conditions that an inductive invariant must assert.

We must now verify conditions I1–I3. I've been writing invariance proofs for many years, and for such a simple protocol I can check these conditions in my head. You may not be so good at it, so let's go through the dull, plodding proof. The trick is to let the math tell us what we must do. This is tiresome for such a simple example. With practice, you'll be able to quickly check the trivial steps of the proof and concentrate on the ones that need careful reasoning. However, you should understand how to write a complete proof before you start cutting corners. For complex algorithms, the only way to prevent errors is by checking all the steps, as tiresome as that may be.

We have three conditions to verify, so we do them one at a time—in any order. Let's check the simplest ones first.

Condition I3 is $Inv \Rightarrow MutualExclusion$. It is obviously true because *MutualExclusion* is a conjunct of *Inv*.

Condition I1 is $Init \Rightarrow Inv$. We can't really prove this, since we don't know what formula *Init* is. However, in writing the protocol's code, I made two implicit assumptions about the initial predicate:

Init1. Variables have values of the proper type.

Init2. Each process is started outside the protocol code.

With these assumptions, we can (informally) prove $Init \Rightarrow Inv$. Since *Inv* is the conjunction of three formulas, we have to prove that *Init* implies each of them. Here is the proof.

Condition I2 is $Inv \wedge Next \Rightarrow Inv'$, where Inv' is *Inv* with all the variables primed. To prove this, we need to know what the next-state action *Next* is. The protocol describes two actions of each process:

$e1(i)$ Describes the execution of statement $e1$.

$e2(i)$ Describes the execution of the **if** test of statement $e2$, which transfers control either to the critical section or to the statement following the **if**.

The next-state action of process i is the disjunction of those two actions plus two others:

$CS(i)$ Describes the execution of process i's critical section. We make the following assumptions about it:

1. It is enabled only when control is in the critical section.
2. It leaves control either in the critical section or outside the protocol.
3. It leaves x unchanged.
4. It does not make *TypeOK* false.

$Rest(i)$ Describes the steps of process i outside the protocol. We make the following assumptions about it:

1. It is enabled only when control is outside the protocol.
2. It leaves control either outside the protocol or at label $e1$.
3. It leaves $x[1-i]$ unchanged.
4. It does not make *TypeOK* false.

Note that we allow the $Rest(i)$ action to change the value of $x[i]$. For example, the algorithm could set $x[i]$ to FALSE to allow the other process to enter its critical section.

The next-state action *Next* then equals

$$\exists\, i \in \{0,1\} : e1(i) \lor e2(i) \lor CS(i) \lor Rest(i)$$

and Condition I3 becomes

$$Inv \land (\exists\, i \in \{0,1\} : e1(i) \lor e2(i) \lor CS(i) \lor Rest(i)) \Rightarrow Inv'$$

The structure of the formula immediately leads to this high-level proof structure. We next prove steps 1–4 separately, in any order. Step 2 is the most interesting, since it's an $e2(i)$ step by which process i enters its critical section. Since Inv' is a conjunction, here is the natural high-level proof of step 2. In writing step 2.3, I used the fact that to prove a formula $\forall j \in S : P(j)$, it suffices to assume $j \in S$ and prove $P(j)$. I substituted j for the bound symbol i in the third conjunct of Inv' to avoid conflict with the symbol i introduced in the statement of step 2.

Let's consider step 2.2. It's simple enough that you can probably see right away why it's true. However, if you're not absolutely sure that it is true, you can write a proof like this one. If you're unsure of the correctness of any of its lowest-level paragraph proofs, you can expand the paragraph proof to a sequence of lower-level steps.

The rest of the proof is similar. For example, here's a proof of step 2.3. The proofs of the high-level steps 3 and 4 use the assumptions made above about the actions $CS(i)$ and $Rest(i)$. You can complete the proof yourself.

Problem 7.2 (a) Write a complete proof of condition I2, the inductive invariance of *Inv*.

(b) Where does the proof of inductive invariance fail if we remove the *TypeOK* conjunct from *Inv*?

This is a boring proof. However, observe that once we discovered the inductive invariant *Inv*, the proof required no insight or creativity. It was just a matter of repeatedly using the structure of the formula to be proved to decompose the proof into simpler steps, until we reach the point where the steps have such simple proofs that it's easy to see they are correct. If you have to go down to such a low level of detail to be confident of the correctness of a proof, you should consider checking the proof with TLAPS□. Of course, that's impossible for this proof without precise definitions of the operators *e*1, *e*2, *CS*, and *Rest*.

Finding an inductive invariant can require creativity. The method of successively strengthening an invariant until it is inductive can guide you. But without insight, it could take quite a few iterations until you find one—perhaps even an infinite number of iterations. The only way to become proficient at finding an inductive invariant is through practice. However, TLC can help you.

7.2.3 Using TLC to Check an Inductive Invariant

As you get better at writing proofs, it becomes increasingly difficult to prove something that isn't true. So, before trying to prove anything, you should first try to use TLC to check if it really is true. We can't use TLC (or any other tool) to check a property of the One-Bit Protocol until we have specified it precisely. Let's specify it as a complete PlusCal algorithm called *OneBitProtocol*.

Other than the implicit variable *pc*, the algorithm uses only the single variable x, where $x[i]$ can initially have either Boolean value, for each process i. There are two processes, named 0 and 1. The algorithm therefore has the following structure, where BOOLEAN is a built-in TLA$^+$ symbol defined to equal the set {TRUE, FALSE} of Booleans.

```
--algorithm OneBitProtocol {
  variable x ∈ [ {0,1} → BOOLEAN ] ;
  process (P ∈ {0,1}) { ... }
}
```

The protocol can be repeated any number of times by a process, so the body of the **process** statement should be:

```
r : while(TRUE)
      {     rest of process code
        e1: x[self] := TRUE ;
        e2: if (¬x[1−self]) { cs: critical section }
      }
```

We now have to decide how to represent the critical section and the *rest of process code*.

The simplest way to represent the critical section is with a **skip** statement, which does nothing except advance the process's control state. This represents

the critical section as a single atomic step, with $InCS(i)$ equal to $pc[i] =$ "cs". We can do this for the same reason that we could represent the *put* and *get* operations as **skip** statements in algorithm *AltSpec*, the specification of alternation in Section 6.9. The **skip** statement is an abstraction that represents all but the last step performed executing the critical section as stuttering steps of the algorithm with $pc[i] =$ "cs".

The *rest of process code* in process *self* is allowed only to change $x[self]$ and $pc[self]$, and it can't jump to $e2$ or cs. Since we don't want to worry about what the other process might do if it reads $x[self]$ to be a non-Boolean value, we want $x[self] \in \text{BOOLEAN}$ to be an invariant. Hence, process *self* should set $x[self]$ only to a Boolean value. Here is a PlusCal statement that sets $x[self]$ to an arbitrarily (nondeterministically) chosen Boolean value:

with $(v \in \text{BOOLEAN}) \{ x[self] := v \}$

Recall that the PlusCal statement **with** $(id \in S) \{ \Sigma \}$ executes the code Σ with an arbitrarily chosen value in the set S substituted for the identifier id. (There can be no labels in Σ.)

The algorithm should allow this **with** statement to be executed any number of times before the process reaches $e1$. We can express this by letting the **while** loop begin:

```
r : while(TRUE)
      { either { with (v ∈ BOOLEAN ) { x[self] := v } ;
                 goto r
               }
        or     skip ;
```

The PlusCal statement

either $\{ \Sigma_1 \}$ **or** $\{ \Sigma_2 \}$ ... **or** $\{ \Sigma_k \}$

executes a nondeterministically chosen Σ_i. (The curly braces are optional for a Σ_i consisting of a single statement.)

Open a new specification *OneBitProtocol* in the Toolbox. It will need to extend the *Integers* module. Insert the ASCII text of the algorithm and run the PlusCal translator on it. Create a new model and run TLC on it. TLC should find 35 reachable states. The TLA$^+$ definitions of the state predicates used in our informal proof are:

ASCII version

$$TypeOK \triangleq \land pc \in [\{0, 1\} \to \{\text{"r"}, \text{"e1"}, \text{"e2"}, \text{"cs"}\}] \\ \land x \in [\{0, 1\} \to \text{BOOLEAN}]$$

$$InCS(i) \triangleq pc[i] = \text{"cs"}$$

$$MutualExclusion \triangleq \neg(InCS(0) \land InCS(1))$$

$$Inv \triangleq \begin{array}{l} \land\ TypeOK \\ \land\ MutualExclusion \\ \land\ \forall\, i \in \{0,\, 1\} : InCS(i) \lor (pc[i] = \text{“e2”}) \Rightarrow x[i] \end{array}$$

Add these definitions to the specification. The first thing we should have TLC check is that *MutualExclusion* is an invariant. We should actually have done this before even trying to write our informal proof. Writing the protocol as a PlusCal algorithm and checking its correctness with TLC is easier than writing even an informal proof. Since TLC can easily check all possible executions of this simple algorithm, there was no need to write any proof. We wrote the proof as an exercise in proof writing, not to check correctness of the protocol. For an N-process mutual exclusion algorithm, TLC can check correctness only for particular values of N—often for values no greater than 3.

Before checking that *Inv* is an inductive invariant, we should check that it is an invariant. This checks that it is true in the initial state (the first of the three conditions in our proof). Of course, TLC does this in milliseconds (plus its startup time) for all executions of this simple algorithm.

We want TLC to check that *Inv* is an inductive invariant of the next-state action *Next* (the second of the three conditions in our proof). Inductive invariance means that if we take a *Next* step starting in any state satisfying *Inv*, we get a state that also satisfies *Inv*. However, TLC can check only ordinary invariance—meaning that *Inv* is true in every state obtained by starting in a state satisfying the initial predicate and taking steps satisfying the next-state action. To check inductive invariance of *Inv*, we consider the specification *ISpec* having initial predicate *Inv* and next-state action *Next*. (We can write *ISpec* in TLA$^+$ as $Inv \land \Box[Next]_{\langle x,\, pc\rangle}$.) The key is:

Question 7.3 Show that *Inv* is an inductive invariant of *Next* iff it is an ordinary invariant of the specification *ISpec*.

Let's check that *Inv* is an invariant of *ISpec*. Create a new model having *Inv* as the initial predicate and *Next* as the next-state action and add *Inv* to the list of invariants to be checked. Alternatively, you can use the temporal-formula specification:

$Inv \land \Box[Next]_{\langle x,\, pc\rangle}$ `Inv /\ [][Next]_<<x, pc>>`

TLC will find that *Inv* is an invariant of this specification, and it will report that there are 35 reachable states. That's the same number of states it found for the original specification, which implies that every state satisfying the inductive invariant *Inv* is reachable. This is not always the case. More often, the inductive invariant allows states not reachable by the algorithm. Some of those unreachable states might be deadlock states, so you should unselect deadlock checking when using TLC to check inductive invariance.

Now change the definition of *Inv* by reversing the order of the conjuncts *TypeOK* and *MutualExclusion*, and run TLC on the specification *ISpec* to check that *Inv* is an inductive invariant of *Next*. TLC reports the error:

```
In evaluation, the identifier pc is either undefined or not an
operator.
```

To understand why this happens, review the description in Section 2.6 of how TLC computes the possible initial states of a spec. It explains why the type-correctness invariant must almost always be the first conjunct of an inductive invariant that you check with TLC.

The way TLC computes the initial states for such a specification implies that it first computes all states satisfying the type-correctness invariant. It then throws away states that don't satisfy the other conjuncts. For most specifications, there are a huge number of type-correct states. TLC can therefore usually check inductive invariance for only very tiny models. If there are too many states satisfying the type-correctness invariant, TLC will report the error:

```
Too many possible next states for the last state in the trace.
```

The largest number of type-correct states that TLC can handle is specified by a parameter called Cardinality of largest enumerable set, which has the default value of one million. You can change its value in the TLC Options section of the Advanced Options model page.

Finding an inductive invariant can be difficult. You'll need all the help that TLC can provide. Even a very tiny model can show that an invariant needs to be strengthened to be inductive. You can often use tricks to reduce the number of states TLC must examine.

As an example, suppose the type invariant simply asserts that p is in the set *Nat* of natural numbers. TLC obviously cannot enumerate all the values of p. A simple solution to this problem is to redefine *Nat* in the Definition Override section of the Advanced Options model page so it equals $0 \,..\, 99$. However, if the algorithm can increment p by 1, then TLC would not find the type invariant to be inductive because it would find $99 + 1$ not to be in *Nat*. To prevent TLC from reporting this error, you can add the State Constraint $p \leq 98$ on the Advanced Options model page. (TLC checks the invariant before checking if the state satisfies the state constraint.)

In this same example, suppose there is another variable q for which the type invariant asserts that q is a natural number but the rest of the invariant implies it is always in $p \,..\, (p + 2)$. You could modify the type invariant by replacing $q \in \mathit{Nat}$ with $q \in p \,..\, (p + 2)$, so TLC has to examine only $100 * 3$ pairs of p, q values rather than $100 * 100$.

Discovering that a predicate is not an inductive invariant by trying to prove that it is can take a lot of time. It's worth putting quite a bit of effort into using

TLC to catch errors. And don't forget that your inductive invariant must be an ordinary invariant of the specification. Whenever you make any changes to it, check first that it's still an invariant.

Question 7.4 (a) Use TLC to show that *TypeOK* is also an inductive invariant of *Next*.

(b) When you do this, or when you run TLC on the specification *ISpec*, TLC reports that the diameter of the state graph is 1. Why?

7.3 The Two-Process One-Bit Algorithm

7.3.1 The Two-Process Algorithm

We now turn the one-bit protocol into a complete mutual exclusion algorithm, starting with a two-process one. We have seen that the protocol ensures mutual exclusion, but we have not discussed deadlock freedom. Our *OneBitProtocol* PlusCal algorithm is obviously not deadlock free because it permits an execution in which $x[0]$ and $x[1]$ both always equal TRUE, so no process ever enters its critical section.

In a mutual exclusion algorithm, we want $x[i]$ to equal FALSE except when process i is in its critical section or trying to enter it. The obvious way to do this is to let each $x[i]$ initially equal FALSE and let the body of process *self* be:

```
ncs: while (TRUE)
       {    skip ;
         e1: x[self] := TRUE ;
         e2: if (¬x[1 − self]) { cs: skip }
             else { goto e2 } ;
         f:  x[self] := FALSE
       }
```

The *ncs* action (execution from label *ncs* to label *e*1) represents the noncritical section. For the same reason we can represent the noncritical section as an atomic **skip** statement, we can also represent the noncritical section as one.

Complete this code to a two-process algorithm named *OneBit*, and put it in a new specification *OneBit2Procs*. The result should look like this. Run the translator on the algorithm. As we did for the protocol specification, define:

$$InCS(i) \triangleq pc[i] = \text{“cs”}$$
$$MutualExclusion \triangleq \neg(InCS(0) \land InCS(1))$$

and let TLC check that *MutualExclusion* is an invariant of the algorithm. You can also check that the predicate *Inv* we defined for the protocol, except with a suitably modified definition of *TypeOK*, is an inductive invariant of the algorithm.

Problem 7.5 Have TLC check that this algorithm *OneBit* implements the algorithm *OneBitProtocol* defined above under the following refinement mapping:

$$\begin{array}{ll} x & \leftarrow \ x \\ pc & \leftarrow \ [i \in \{0,1\} \mapsto \text{IF } pc[i] \in \{\text{“ncs”}, \text{“f”}\} \text{ THEN “r”} \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } pc[i]\,] \end{array}$$

(See Section 6.6□.)

Of course, we expected the algorithm to satisfy mutual exclusion since the protocol does. However, we want an algorithm that also is deadlock free. Deadlock freedom was Dijkstra's requirement (d). Today, that requirement would be stated as:

> If any process tries to enter its critical section, then some process eventually reaches its critical section.

To state this more precisely, we first define a state predicate $Trying(i)$ that is true iff process i is trying to enter its critical section. The definition is

$$Trying(i) \ \triangleq \ pc[i] \in \{\text{“e1”}, \text{“e2”}\}$$

Add the definitions of *Trying* and *DeadlockFree* to the specification.

Deadlock freedom means that if $Trying(0) \lor Trying(1)$ ever becomes true, then eventually (at that point or some later point in the execution) $InCS(0) \lor InCS(1)$ will be true. This assertion is written as the temporal formula

$$DeadlockFree \ \triangleq \ (Trying(0) \lor Trying(1)) \rightsquigarrow (InCS(0) \lor InCS(1))$$

In general, the temporal operator $\rightsquigarrow$ (read *leads to* and written in ASCII as `~>`) is defined so that for any state predicates P and Q, the formula $P \rightsquigarrow Q$ is true of a behavior $s_1 \to s_2 \to \ldots$ iff, for any i such that P is true in state s_i, there is a $j \geq i$ such that Q is true in state s_j.

The translation defines the specification *Spec* of the algorithm to be $Init \land \Box[Next]_{vars}$. This specification can't satisfy deadlock freedom because it specifies only safety; it doesn't require the algorithm to take any (non-stuttering) steps. Thus, it allows a behavior in which process 0 remains forever in its noncritical section and process 1 reaches control point $e2$ and stops. We must add some fairness assumption.

The traditional fairness assumption for multiprocess algorithms, and the one implicitly assumed by Dijkstra, is weak fairness□ of each process—that is, of each process's next-state action. As we saw in our specification of alternation, this assumption is specified in PlusCal by preceding the keyword **process** with the keyword **fair**. Make this change to the algorithm and run the translator. The translator then adds the conjunct

$$\forall self \in \{0,1\} : WF_{vars}(P(self))$$

to specification *Spec*, where $P(i)$ is the next-state action of process i.

Have TLC check the property *DeadlockFree* (by adding it to the Properties list in the What to check? section of the model's Model Overview page). TLC reports that the property is not satisfied; it gives an error trace that reaches the state with $x[0]$ and $x[1]$ both equal to TRUE and $pc[0]$ and $pc[1]$ both equal to "e2", and then stutters forever. From this state, all the algorithm can do is have either process i execute its $e2$ action, finding $x[1-i]$ equal to TRUE and remaining in the same state.

Question 7.6 Explain why $\mathrm{WF}_{vars}(Proc(i))$ is true for a behavior in which eventually $x[1-i] \wedge (pc[i] = \text{"e2"})$ is always true. ANSWER

To make the algorithm deadlock free, we must prevent both processes from waiting forever at $e2$. The One-Bit algorithm does this by having one process i set $x[i]$ to FALSE and allowing the other process to enter its critical section. Let process 1 be the one to do that. We leave process 0 the same and replace the **else** clause of process 1 with

```
e3: x[1] := FALSE ;
e4: while (x[0]) { skip} ;
    goto e1
```

The two processes now don't execute the same code. We could declare them with two separate **process** statements, but it's more convenient to use a single **process** statement, replacing statement $e2$ with:

ASCII version

```
e2: if (¬x[1 − self]) { cs: skip }
    else { if (self = 0) { goto e2 }
           else { e3: x[1] := FALSE ;
                  e4: while (x[0]) { skip} ;
                      goto e1
                }
         }
```

Change the algorithm in module *OneBit2Procs*. The definition of *Trying* also needs to be changed to reflect the change to the code. A suitable definition is

$$Trying(i) \triangleq pc[i] \in \{\text{"e1"}, \text{"e2"}, \text{"e3"}, \text{"e4"}\}$$

Since $pc[0]$ does not equal "e3" or "e4" in any reachable state, we could also define *Trying* by

$$Trying(i) \triangleq pc[i] \in \text{IF } i = 0 \text{ THEN } \{\text{"e1"}, \text{"e2"}\} \text{ ELSE } \{\text{"e1"}, \text{"e2"}, \text{"e3"}, \text{"e4"}\}$$

but the simpler definition will do just as well. Run the translator and run TLC on the same model as before. TLC should verify that this algorithm does satisfy property *DeadlockFree*.

Problem 7.7 Find an inductive invariant of the algorithm that can be used to prove the invariance of *MutualExclusion*. Write the invariance proof. ANSWER

7.3.2 Busy Waiting Versus Synchronization Primitives

Consider the code

$e4$: **while** $(x[0])$ { **skip**} ;
goto $e1$

executed by process 1. This is the way Dijkstra might have written that code, indicating that the process keeps reading $x[0]$ until finding it equal to FALSE, whereupon it goes to location $e1$. Since Dijkstra posited reading and writing shared memory registers as the only synchronization primitives, a process could wait for $x[0]$ to become false only by repeatedly reading it.

A more natural way to write this code in PlusCal is with an **await** statement:

$e4$: **await** $\neg x[0]$;
goto $e1$

In 1965, the two versions of $e4$ would have been considered to produce two different algorithms. The **await** construct would have been viewed as a special synchronization primitive, very different from the **while** loop of the first version. Most computer scientists today would probably also consider them to be different. However, the two PlusCal algorithms are completely equivalent.

To see that the two versions of the code produce equivalent specs, we need only examine their TLA$^+$ translations. The translation of the algorithm is the formula *Spec*, defined to equal

$$Init \ \land \ \Box[Next]_{vars} \ \land \ (\forall self \in \{0, 1\} : WF_{vars}(P(self)))$$

where

Init is the initial predicate.

Next is the algorithm's next-state action.

$P(self)$ is the next-state action of process *self*.

vars is the pair $\langle x, pc \rangle$

In the version with the **while** loop, $e4$ is defined by:

$$\begin{array}{lll} e4(self) \triangleq & \land\, pc[self] = \text{“e4”} & \\ & \land \text{ IF } x[0] & \\ & \quad \text{THEN} & \land \text{ TRUE} \\ & & \quad \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“e4”}] \\ & \quad \text{ELSE} & \land\, pc' = [pc \text{ EXCEPT } ![self] = \text{“e1”}] \\ & \land\, x' = x & \end{array}$$

In the version with **await**, $e4$ is defined by:

$$\begin{aligned} e4(self) \triangleq\ & \land\ pc[self] = \text{“e4”} \\ & \land\ (\neg x[0]) \\ & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“e1”}] \\ & \land\ x' = x \end{aligned}$$

When $pc[self]$ equals “e4” and $x[0]$ equals TRUE, the first definition allows an $e4(self)$ step that leaves pc and x unchanged—that is, a stuttering step. (Setting the new value of $pc[self]$ to equal its old value is the same as not changing it.) With those values of $pc[self]$ and $x[0]$, the second definition does not allow an $e4(self)$ step. Hence the definitions of $Next$ and $P(self)$ differ only in whether or not they allow certain stuttering steps (ones that leave x and pc unchanged). Since action $[Next]_{vars}$ allows stuttering steps, the two definitions of $e2(self)$ yield equivalent definitions of $[Next]_{vars}$. They also yield equivalent definitions of $\langle P(self)\rangle_{vars}$, which allows only the non-stuttering steps allowed by $P(self)$. Hence, the definition of WF implies that they yield equivalent definitions of $\text{WF}_{vars}(P(self))$. The two definitions therefore yield equivalent definitions of the specification $Spec$.

Problem 7.8 Use TLC as follows to show that the two specifications $Spec$ are equivalent. Put the two versions of the algorithm in two different specifications, with root modules that I will call here $M1$ and $M2$. Add the following statement to $M1$

$$Other \triangleq \text{INSTANCE } M2 \text{ WITH } x \leftarrow x, pc \leftarrow pc$$

and use TLC to check that the algorithm of $M1$ satisfies the property $Other!Spec$. Explain why this shows that formula $Spec$ of $M1$ implies formula $Spec$ of $M2$. Then use the analogous procedure to show that formula $Spec$ of $M2$ implies formula $Spec$ of $M1$, showing that the two formulas are equivalent.

The equivalence of the two algorithms doesn't mean that busy waiting is the same as using an operating-system primitive to wait for a condition to be true. What it does mean is that the PlusCal code describes the algorithm at a high enough level of abstraction that the distinction between these two ways of waiting disappears. In the algorithm's abstraction, waiting means not changing pc or x. This describes implementations in which a waiting process repeatedly reads the value of $x[0]$, as well as implementations in which a waiting process “sleeps” until it is notified that $x[0]$ has been changed.

PlusCal is not a programming language. It is a convenient way to write certain kinds of system specifications—including ones that are usually called “algorithms”. Like any system specification, an algorithm is not a system; it is a mathematical formula that serves as a blueprint of a system. What matters is the mathematical formula, not the syntax of the PlusCal code that generated it.

7.3.3 Requirement (c)

TLC has checked that the two-process One-Bit algorithm satisfies deadlock freedom, which is the precise statement of Dijkstra's requirement (d). We have decided to ignore his requirement (a), which was symmetry of the processes' code. His requirement (b), that there is no assumption about the relative speeds, is built into our method of specifying algorithms. (Any such assumption would have to be explicitly asserted.) What about his requirement (c)?

As we observed above, requirement (c) implicitly requires that either process be able to stop in its noncritical section. In our algorithm, a process i is in its noncritical section iff it is at control point *ncs*—that is, iff $pc[i] =$ "ncs". Our liveness requirement of weak fairness on the next-state action $P(i)$ of each process i means that the process cannot stop taking non-stuttering steps if such a step remains enabled. When $pc[i] =$ "ncs", the process can execute the **skip** statement, which is a non-stuttering step (it sets $pc[i]$ to "e1"). Hence weak fairness of $P(i)$ implies that process i cannot stop in its noncritical section, violating requirement (c).

As we saw for the finer-grained *Handshake* algorithm□, PlusCal allows us to modify the fairness requirement so that it does not apply to a process when control is at *ncs* by putting `-` after the label, like this:

ncs :- **while** (TRUE) ...

This causes the translation to produce the fairness assumption:

$$\forall\, self \in \{0, 1\} : \mathrm{WF}_{vars}((pc[self] \neq \text{"ncs"}) \wedge P(self))$$

Weak fairness of the action $(pc[i] \neq \text{"ncs"}) \wedge P(i)$ requires process i eventually to take a $P(i)$ step only when $pc[i]$ does not equal "ncs", so it allows the process to stop in its noncritical section. Have TLC check that the algorithm still satisfies deadlock freedom with this weaker fairness assumption.

It's not clear whether a process should be allowed to remain forever in its critical section. Dijkstra's article says nothing about this possibility, and I don't know if he considered it. Most computer scientists seem to assume that a process must eventually leave its critical section. Indeed, my statement of starvation freedom—that any process trying to enter its critical section eventually succeeds—makes it impossible to satisfy if a process can remain forever in its critical section. However, it can be argued that the only assumption a mutual exclusion algorithm should make about the noncritical and critical sections is that they do not modify the values of any variables used by the algorithm—except for the obvious changes to the value of *pc*.

For deadlock freedom, it makes little difference whether or not we allow a process to remain forever in its critical section. However, assuming that it can't allows a simpler definition of starvation freedom. I will therefore make the customary assumption that a process in its critical section eventually exits from it.

Question 7.9 What is the weak fairness property that allows processes to stop inside their critical and noncritical sections? Compare your answer to the fairness condition produced by the translator when you modify the PlusCal code to allow this possibility. Use TLC to check that deadlock freedom is satisfied even with this weaker fairness assumption.

Question 7.10 Assuming the appropriate definitions of *Trying* and *InCS*, write the fairness formula expressing the customary statement of starvation freedom for an algorithm with an arbitrary set *Proc* of processes. ANSWER

7.4 Proving Liveness

7.5 An Informal Proof

In Question 7.7, you proved the safety property of the Two-Process One-Bit Algorithm—namely, that it satisfies mutual exclusion. Let's now prove liveness. The liveness property satisfied by this algorithm is deadlock freedom. Letting $Trying(i)$ mean that process i is trying to enter its critical section and $InCS(i)$ to mean that it is in its critical section, we expressed this condition above by the formula

$$DeadlockFree \triangleq (Trying(0) \lor Trying(1)) \rightsquigarrow (InCS(0) \lor InCS(1))$$

For our algorithm, we have

$$Trying(i) \triangleq pc[i] \in \{\text{“e1”}, \text{“e2”}, \text{“e3”}, \text{“e4”}\}$$
$$InCS(i) \triangleq pc[i] = \text{“cs”}$$

(The algorithm's invariant implies that $pc[0]$ never equals “e3” or “e4”.) We now give an informal proof of this property. We must prove that it is true for some arbitrary behavior $s_1 \to s_2 \to s_3 \to \cdots$ satisfying the algorithm's specification. For linguistic convenience, we say that something is true at time t if it is true in state s_t or in the suffix $s_t \to s_{t+1} \to \cdots$ of this behavior.

For brevity, let's define

$$T0 \triangleq Trying(0) \qquad T1 \triangleq Trying(1) \qquad Success \triangleq InCS(0) \lor InCS(1)$$

We have to show that if $T0 \lor T1$ is true at some time t_1, then *Success* is true at some time $t \geq t_1$. In general, proving a formula F by contradiction is easier than proving it directly because we have the additional hypothesis $\neg F$. To prove by contradiction that a formula F is eventually true, we get to assume not just that $\neg F$ is true, but that it is always true. This makes proof by contradiction especially useful. We are therefore led to the following high-level proof.

We can prove F by assuming $\neg F$ and proving F, since $\neg F \land F$ implies FALSE.

1. It suffices to assume that $T0 \lor T1$ is true at some time t_1 and $\neg Success$ is true at all times $t \geq t_1$, and to obtain a contradiction.
2. $T0$ is false at time t_1.
3. $T1$ is false at time t_1.
4. Q.E.D.

   PROOF: By 2, 3, and the step 1 assumption.

When writing the proof of step 3, we discover that we need to know that $T0$ is false at some time $t_2 \geq t_1$. We can avoid a separate proof of this fact by strengthening step 2 to:

2. $T0$ is false at all times $t > t_1$.

Here is the algorithm and here is the proof, carried down to a reasonable level of detail. I have omitted the proof of step 2.

If you are not interested in writing more rigorous proofs, skip to Section 8. Otherwise, detour to a discussion of temporal logic before continuing to the next section.

7.6 A More Formal Proof

We now give a more formal version our informal proof of deadlock freedom—a proof in which each assertion is a TLA$^+$ formula. For convenience in writing the proof, let's define *Fairness* to be the formula expressing the fairness property of the algorithm, so *Spec* equals $Init \land \Box[Next]_{vars} \land Fairness$. The TLA$^+$ translation of the algorithm tells us that its definition is:

$$Fairness \triangleq \forall\, self \in \{0, 1\} : \mathrm{WF}_{vars}((pc[self] \notin \{\text{“ncs”}, \text{“cs”}\}) \land P(self))$$

However, we won't carry our proof down to the level of detail for which the formal definition of *Fairness* matters. The only formal property of *Fairness* that we need to know is that it's a $\Box$ formula. This follows from the fact that the definition of WF implies that any WF formula is a $\Box$ formula. However, you should be able to verify intuitively from the meaning of this fairness condition that *Fairness* is true of a behavior σ iff it is true of all suffixes of τ, which implies that it is a $\Box$ formula.

To prove that the algorithm is deadlock free, we must prove the theorem $Spec \Rightarrow DeadlockFree$. The usual way to prove such an implication is to assume *Spec* and prove *DeadlockFree*. However, this is a temporal theorem, and assumptions in temporal proofs should be $\Box$ formulas. Formula *Spec* is not a $\Box$ formula because of its conjunct *Init*, which is a state predicate.

We should not expect assuming *Spec* to be useful. To prove that an arbitrary behavior σ satisfying *Spec* satisfies a liveness property, we reason about states that occur at arbitrary points in the behavior. In other words, we reason about

suffixes of σ. The formula *Spec* is true of σ but not necessarily of a suffix of σ, because the initial predicate *Init* is not true for an arbitrary state in σ. Therefore, the fact that *Spec* is true of σ cannot be used directly to reason about suffixes of σ.

To prove a liveness property, instead of using *Init*, we need to use a state predicate that is always true—in other words, we must use an invariant. Let's call the invariant *LInv*. Our proof begins as follows, where we first prove that *LInv* is an invariant and then use $\Box LInv$ as an assumption.

1. $Spec \Rightarrow \Box LInv$
2. SUFFICES ASSUME: $\Box LInv \land \Box[Next]_{vars} \land Fairness$
   PROVE: $DeadlockFree$

The invariant *Inv* that we used to prove mutual exclusion is not strong enough for proving deadlock freedom. For example, step 3.3 of our informal proof uses the fact that $x[0]$ equals FALSE when process 0 is at *ncs*. However, that fact was not needed to prove mutual exclusion and is not implied by *Inv*. Here is an invariant that is strong enough. It strengthens *Inv* by specifying the value of $x[i]$ as a function of $pc[i]$, for each process i.

$$\begin{aligned} LInv \triangleq\ & \land\ TypeOK \\ & \land\ MutualExclusion \\ & \land\ pc[0] \notin \{\text{“e3”}, \text{“e4”}\} \\ & \land\ \forall\, i \in \{0, 1\} : x[i] \equiv (pc[i] \in \{\text{“e2”}, \text{“e3”}, \text{“cs”}, \text{“f”}\}) \end{aligned}$$

Since *DeadlockFree* equals $T0 \lor T1 \leadsto Success$, we use a standard temporal proof by contradiction with the next proof step:

3. SUFFICES ASSUME: $\Box\neg Success$
   PROVE: $(T0 \lor T1) \leadsto$ FALSE

When you get used to writing these proofs, you will save space by combining steps 2 and 3 into:

2. SUFFICES ASSUME: $\Box LInv \land \Box[Next]_{vars} \land Fairness \land \Box\neg Success$
   PROVE: $(T0 \lor T1) \leadsto$ FALSE

Here is the full proof. If you'd like to read the proof in the Toolbox, using its commands for hiding subproofs, click here for an ASCII version. (It does not contain the graphs that explain the uses of Leads-To Induction.)

7.7 The *N*-Process One-Bit Algorithm

It's easy to generalize the basic 2-process one-bit protocol to N processes. Each process i sets its bit $x[i]$ to TRUE and can enter its critical section if it then sees $x[j]$ equal to FALSE for every other process j. We first write this protocol in pseudo-PlusCal.

Let *Procs* be the set of processes, which following Dijkstra we take to be the set $1 \,..\, N$ of integers from 1 through N. As in PlusCal, we let *self* be the name of the current process. We let process *self* read the other bits $x[j]$ in an arbitrary, nondeterministically chosen order. We let *unchecked* be a variable local to process *self* that holds the set of all processes j for which *self* has not yet read $x[j]$. Process *self* sets *unchecked* to the set of all processes in *Procs* other than *self* itself—that is, the set $Procs \setminus \{self\}$. It then repeatedly sets a local variable *other* to any process in the set *unchecked*, removes *other* from *unchecked*, and then continues only if $x[other]$ is FALSE. The pseudo-code for the protocol is:

```
e1: x[self] := TRUE ;
    unchecked := Procs \ {self} ;
e2: while(unchecked ≠ {})
      { Set other to any element of unchecked ;
        unchecked := unchecked \ {other} ;
        await ¬x[other]
      }
cs: critical section
```

As we have seen in the case $N = 2$, this algorithm can deadlock—each process waiting for the other's variable to become false. The two-process algorithm breaks this deadlock by having process 1 set its variable false, allowing process 0 to enter the critical section. The generalization to N processes is to have each process wait for any lower-numbered process that it observes also to be waiting to enter the critical section. Here is the algorithm in PlusCal, where *Procs* is defined to be the set $1 \,..\, N$ of processes. As in the two-process version, we assume fairness of each process, but not of its non-critical section action.

Create a new specification with the ASCII version of the algorithm and run the translator. (You will have to declare N to be a constant and define *Procs*.) Use TLC to check that the algorithm satisfies mutual exclusion by checking that

$$MutualExclusion \triangleq \forall\, i, j \in Procs : (i \neq j) \Rightarrow \neg(InCS(i) \land InCS(j))$$

is an invariant, and check that the algorithm is deadlock free by checking that it satisfies the property

$$DeadlockFree \triangleq (\exists\, i \in Procs : Trying(i)) \rightsquigarrow (\exists i \in Procs : InCS(i))$$

where *Trying* and *InCS* are defined by:

$$Trying(i) \triangleq pc[i] \in \{\text{“e1”}, \text{“e2”}, \text{“e3”}, \text{“e4”}, \text{“e5”}, \text{“e6”}\}$$

$$InCS(i) \triangleq pc[i] = \text{“cs”}$$

TLC will check these properties in a few seconds for $N = 3$ and in around 20 minutes for $N = 4$.

To prove mutual exclusion, we first define the state predicate $Past(i, j)$ to assert that process i is trying to enter its critical section and has "passed" process j, meaning that it has seen $x[j]$ equal to FALSE when executing the **if** $(x[other])$ test of statement $e3$. The precise definition is:

In the TLA$^+$ translation, the process-local variables *unchecked* and *other* becomes arrays indexed by *Proc*.

$$
\begin{aligned}
Past(i,\, j) \;\triangleq\; & \lor\, (pc[i] = \text{“e2”}) \land (j \notin unchecked[i]) \\
& \lor\, \land\, pc[i] \in \{\text{“e3”},\ \text{“e6”}\} \\
& \quad \land\, j \notin unchecked[i] \cup \{other[i]\} \\
& \lor\, pc[i] = \text{“cs”}
\end{aligned}
$$

The basic reason the algorithm achieves mutual exclusion is that, when $Past(i, j)$ is true, $Past(j, i)$ cannot become true because $x[i]$ is true. In other words, the following formula is an invariant of the algorithm:

$$\forall\, i \in Procs : \forall\, j \in Procs \setminus \{i\} : Past(i, j) \Rightarrow \neg Past(j, i) \land x[i]$$

Use TLC to check that this is an invariant.

An inductive invariant should contain a type-correctness invariant, which I like to name *TypeOK*, as a conjunct. However, the conjunction of *TypeOK* and the formula above isn't an inductive invariant. If you try writing a proof, you'll quickly discover why it isn't. However, for fun, let TLC show you it isn't. Use the method described above to have TLC check if this invariant is inductive. (Use $N = 2$.) Examine the error trace and figure out why it isn't inductive. (Don't peek at what comes below.)

The error trace reveals that the invariant allows $Past(i, j)$ to be true when $x[i]$ is false, which allows a step to make $Past(j, i)$ also true. We need to add a conjunct asserting that $x[i]$ is true at those points in the code where it obviously is true. This leads us to the following invariant:

$$
\begin{aligned}
Inv \;\triangleq\; & \land\, TypeOK \\
& \land\, \forall\, i \in Procs : \\
& \quad \land\, (pc[i] \in \{\text{“e2”},\ \text{“e3”},\ \text{“e6”},\ \text{“cs”}\}) \Rightarrow x[i] \\
& \quad \land\, \forall\, j \in Procs \setminus \{i\} : Past(i,\, j) \Rightarrow \neg Past(j,\, i) \land x[i]
\end{aligned}
$$

ASCII version

Use TLC to check that this is an inductive invariant for $N = 2$.

Question 7.11 How many states must TLC generate in the course of checking that *Inv* is an inductive invariant for $N = 3$?

ANSWER

Question 7.12 Write a proof that *Inv* is an inductive invariant of the algorithm, and that it implies *MutualExclusion*.

Let's now show that the algorithm is deadlock free. For that, it suffices to assume that, at some time during the execution, some process is trying to enter its critical section but no process ever does, and to obtain a contradiction. A naive argument goes as follows: Consider the smallest i for which $Trying(i)$ ever becomes true. Since any non-trying process j eventually sets $x[j]$ false, eventually process i never reads $x[j]$ true for any j less than i, so it never sets $x[i]$ false. Therefore, every other waiting process must eventually reach $e5$ and wait forever for $x[i]$ to become false. At this point, $x[j]$ is false for all $j \neq i$, so process i must eventually enter the critical section, which is the required contradiction.

This kind of reasoning about executions is unreliable; it's easy to miss a possible sequence of actions. In fact, the argument above is wrong because it ignores the possibility that process i is waiting for $x[j]$ to become false, for some $j > i$, but j is one of a group of processes that keep continually looping from $e1$ to $e5$ and back. Process j keeps setting $x[j]$ alternately false and true, and process i is unlucky and keeps reading $x[j]$ when it is true, remaining forever at $e6$. We need a more rigorous proof.

A rigorous liveness proof needs an invariant. Once again, the invariant Inv used to prove mutual exclusion isn't strong enough to prove deadlock freedom because it asserts when $x[i]$ must be true, but not when it must be false. An inductive invariant that does the job is:

$$
\begin{array}{lll}
LInv & \triangleq & \land\ Inv \\
 & & \land\ \forall\, i \in Procs : \\
 & & \quad \land\ i \notin unchecked[i] \\
 & & \quad \land\ (pc[i] \in \{\text{“ncs”}, \text{“e5”}\}) \Rightarrow \neg x[i] \\
 & & \quad \land\ (pc[i] = \text{“e3”}) \Rightarrow (other[i] \neq i) \\
 & & \quad \land\ (pc[i] \in \{\text{“e4”}, \text{“e5”}\}) \Rightarrow (i > other[i]) \\
 & & \quad \land\ (pc[i] = \text{“e6”}) \Rightarrow (other[i] > i)
\end{array}
$$

Question 7.13 Use TLC to check that $LInv$ is an inductive invariant of the algorithm, then prove that it is

Here is a more rigorous proof of deadlock freedom.

Question 7.14 Make this liveness proof more rigorous by expanding the proof sketches of steps 7–9 into another level of structured proof.

7.8 The Bakery Algorithm

Dijkstra's 1965 paper inspired the publication of many mutual exclusion algorithms. The first, published four months later by Harris Hyman, was incorrect. Four months after that, Donald Knuth published an article pointing out the

error in Hyman's algorithm and presenting the first starvation-free mutual exclusion algorithm. Since then, there have probably been hundreds of published mutual exclusion algorithms. My favorite is called the *bakery algorithm*. It's my favorite because it's the first one that I invented, because it's simple, and because it has a remarkable property that I'll discuss later.

The inspiration for the bakery algorithm comes from a common method for serving customers that I first saw as a child in a neighborhood bakery. Each arriving customer gets a numbered ticket from a dispenser, tickets being numbered successively. The waiting customer with the lowest numbered ticket is the next one served.

In the bakery algorithm, each process that wants to enter its critical section obtains a number, and the process with the lowest number enters its critical section. In keeping with the metaphor, we say that a process that is *not* in its non-critical section is in the bakery. Upon entering the bakery, a process obtains its number by reading the numbers of all other processes in the bakery and setting its own number to a number higher than any that it reads. (The obvious such number is one greater than the highest number it reads, but any larger number also works.) Processes outside the bakery have their numbers equal to 0, so a process entering the bakery simply sets its number to be greater than that of any other process, inside or outside the bakery.

The obvious problem with this approach is that two processes entering the bakery at the same time can choose the same number. This problem is easily solved by naming the processes with numbers and using process names to break ties: if two processes choose the same number, the one with the smallest name enters its critical section first.

7.8.1 The Big-Step Algorithm

We first write a version of the bakery algorithm called the *big-step* algorithm in which a process's entire operation of choosing its number is a single step. Having a process read every other process's number and set its own number all in a single step makes the algorithm rather uninteresting. In fact, it makes it impossible for two different processes ever to have the same non-zero number. However, to make the transition to the finer-grained algorithm easier, we'll ignore that and write the big-step algorithm as if it were necessary to use process names to break ties.

The Algorithm

We start by declaring the number N of processes and defining *Procs* to be the set of processes—more precisely, the set of process names, which are numbers from 1 through N.

MODULE *BigStepBakery*

EXTENDS *Integers*

CONSTANT N
ASSUME $N \in Nat$

$Procs \triangleq 1 \,..\, N$

The algorithm uses an array num, where $num[p]$ is process p's number. We define $\prec$ on pairs of integers so that, if processes p and q are in the bakery, then p enters its critical section before q does iff $\langle num[p], p \rangle \prec \langle num[q], q \rangle$. This is the case iff either $num[p]$ is less than $num[q]$ or else they are equal and $p < q$. The TLA$^+$ definition of $\prec$ is

$$a \prec b \triangleq \lor a[1] < b[1] \\ \lor (a[1] = b[1]) \land (a[2] < b[2])$$

The relation $\prec$ is called the *lexicographical ordering* on the set of pairs of numbers. It is a total ordering on pairs of numbers, meaning that for any pairs A, B, and C of numbers:

- $A \prec B$ and $B \prec C$ imply $A \prec C$.
- Exactly one of the relations $A \prec B$, $B \prec A$, or $A = B$ holds.

Upon entering the bakery, process p can set $num[p]$ to any natural number that is greater than $num[q]$ for every other process q. The set of all such numbers is:

$$\{j \in Nat : \forall q \in Procs \setminus \{p\} : j > num[q]\}$$

Since $num[p]$ equals 0 at that point, this expression can be simplified a bit to:

$$\{j \in Nat : \forall q \in Procs : j > num[q]\}$$

Here is the PlusCal code. In addition to the variable num, it declares the process-local variable *unchecked*. A process uses *unchecked* to store the set of other processes that it has determined it does not have to wait for. To simplify the type-correctness invariant, *unchecked* is initialized to the empty set even though its initial value is never used. Here are what the algorithm's atomic actions do:

- The actions corresponding to the labels *cs* and *ncs* represent the critical and non-critical sections, respectively.
- The *enter* action sets $num[self]$ to an arbitrary integer greater than $num[i]$ for all processes i. It also initializes *unchecked* to the set of all processes except *self*.
- The *wait* statement's **while** loop chooses an arbitrary process i in *unchecked* and, if $num[i]$ equals 0 (so i is not in the bakery) or $\langle num[self], self \rangle$ precedes $\langle num[i], i \rangle$ in the lexicographical ordering (so process *self* should enter its critical section before process i does), then it removes i from *unchecked*. The loop terminates and *self* enters its critical section when *unchecked* is empty.

- After leaving the critical section, the process executes the *exit* statement to set $num[self]$ to 0 and enters the non-critical section.

Safety

The type-correctness invariant and the invariant asserting the mutual exclusion property are:

$$\begin{aligned} TypeOK \triangleq\ & \land num \in [Procs \to Nat] \\ & \land unchecked \in [Procs \to \text{SUBSET } Procs] \\ & \land pc \in [Procs \to \{\text{“ncs”}, \text{“enter”}, \text{“wait”}, \text{“cs”}, \text{“exit”}\}] \end{aligned}$$

$$\begin{aligned} & MutualExclusion \triangleq \\ & \quad \forall\, p,\, q \in Procs : (p \neq q) \Rightarrow \neg((pc[p] = \text{“cs”}) \land (pc[q] = \text{“cs”})) \end{aligned}$$

Here is the ASCII version of a module containing the algorithm and the TLA$^+$ declarations and definitions. Use it to create a new specification in the Toolbox. It will report undefined operator errors until you run the PlusCal translator.

To check the algorithm with TLC, you will have to use a model that redefines *Nat* to be a finite set of numbers. For the assumption $N \in Nat$ and type correctness to hold, *Nat* must contain 0 and N. A model with N equal to 3 and *Nat* defined to equal $0 \ldots 5$ has only 2528 reachable states, and TLC quickly checks that it satisfies the two invariants. TLC finds no errors on the small models it can check quickly. While we let it run on a larger model, it's time to prove that the algorithm satisfies mutual exclusion.

You will probably find the following argument the most intuitively appealing. Suppose process p has entered the bakery and set $num[p]$. Any process q that then enters the bakery sets $num[q]$ greater than $num[p]$, and therefore q cannot enter the critical section while p is still in the bakery. Two processes therefore can't be in their critical sections at the same time, since one of them must have set its number after the other did. To make this proof more rigorous, we must recast it in terms of an invariant.

Mutual exclusion clearly depends on the invariance of:

> **NumPos** If a process p is at its *wait* statement or critical section, then $num[p] > 0$ holds.

Liveness also depends on the fact that $num[p] = 0$ when p is in its noncritical section. To avoid having two separate invariants for safety and liveness, we strengthen NumPos to

> **NumPos** $num[p] > 0$ iff process p is at its *wait* statement, its critical section, or its *exit* statement.

To write the more interesting part of the invariant, let's define:

$$\begin{aligned} Before(p,\, q) \triangleq\ & \lor num[q] = 0 \\ & \lor \langle num[p],\, p\rangle \prec \langle num[q],\, q\rangle \end{aligned}$$

(For the proof of the big-step algorithm, we could replace the second disjunct by $num[p] < num[q]$.) The key invariant is:

> **Before** If a process p is either at the *wait* statement and has executed the **while** loop iteration for process q, or is in its critical section, then $Before(p, q)$ is true.

The NumPos and Before invariants imply mutual exclusion because:

1. It suffices to assume two different processes p and q are in their critical sections and obtain a contradiction.
   PROOF: Obvious.
2. $Before(p, q) \ \land \ Before(q, p)$
   PROOF: By 1 and invariant Before.
3. $(num[p] > 0) \ \land \ (num[q] > 0)$
   PROOF: By 1 and invariant NumPos.
4. $(\langle num[p], p\rangle \prec \langle num[q], q\rangle) \ \land \ (\langle num[q], q\rangle \prec \langle num[p], p\rangle)$
   PROOF: By 2, 3, and the definition of *Before*.
5. Q.E.D.
   PROOF: Since $\prec$ is a total ordering on the set of pairs of integers, 4 is impossible.

The conjunction of invariants NumPos and Before and the type-correctness invariants is an inductive invariant. To define it precisely, we observe that the set $unchecked[p]$ contains the processes for which p has not yet executed the **while** loop body.

Remember that $unchecked[p]$ is p's "copy" of the local variable *unchecked*.

$$
\begin{array}{lll}
Inv & \triangleq & \land \ TypeOK \\
 & & \land \forall\, p \in Procs : \\
 & & \quad \land (pc[p] \in \{\text{“wait”}, \text{“cs”}, \text{“exit”}\}) \equiv (num[p] > 0) \\
 & & \quad \land \forall\, q \in (Procs \setminus \{p\}) : \\
 & & \qquad \lor (pc[p] = \text{“wait”}) \land (q \notin unchecked[p]) \\
 & & \qquad \lor pc[p] = \text{“cs”} \\
 & & \qquad \Rightarrow Before(p,\ q)
\end{array}
$$

ASCII version

Problem 7.15 (a) Using the method described in Section 7.2.3, check that *Inv* is an inductive invariant of the big-step algorithm.

(b) Prove that *Inv* is an inductive invariant of the algorithm, completing the proof that the big-step algorithm satisfies mutual exclusion.

Liveness

Let us now check that the big-step bakery algorithm is starvation free. First, we need to add a fairness assumption for the algorithm. As usual, we assume fairness

for each process by adding the keyword **fair** before the keyword **process** in the PlusCal code. As we observed in Section 7.3.3, satisfying Dijkstra's requirement (c) means we must allow a process to stop in its non-critical section, which we do by putting "-" after the label "`ncs:`". Make those changes and re-translate the algorithm.

Recall the definition of *DeadlockFree* given above for the one-bit algorithm. Modifying the definitions for the big-step bakery algorithm yields:

ASCII version

$$
\begin{aligned}
Trying(p) &\triangleq pc[p] \in \{\text{“enter”}, \text{“wait”}\} \\
InCS(p) &\triangleq pc[p] = \text{“cs”} \\
DeadlockFree &\triangleq (\exists\, p \in Procs : Trying(p)) \rightsquigarrow (\exists\, p \in Procs : InCS(p))
\end{aligned}
$$

Add those definitions to the module and have TLC check the property *DeadlockFree*. It should succeed.

In Question 7.10, you defined starvation freedom by:

$$StarvationFree \triangleq \forall\, p \in Procs : Trying(p) \rightsquigarrow InCS(p)$$

(We are using the simpler definition, which assumes that a process may not stop in its critical section.) Add that definition to the model and have TLC check that the algorithm satisfies it. TLC reports that the property is not satisfied! Have we made a mistake? Before reading further, examine the error trace and see if you can figure out what happened.

When I check starvation freedom for the model with $N \leftarrow 3$ and $Nat \leftarrow 0\,..\,5$, TLC produces an error trace that ends with processes 1 and 2 always at statement *enter*, and process 3 forever repeating the following sequence of steps:

- Enter the bakery.
- Set $num[3]$ to 5.
- Execute the *wait* loop, seeing $num[1]$ and $num[2]$ equal to 0, and enter the critical section.
- Exit the critical section and return to the non-critical section.

This behavior is not allowed by the algorithm. Processes 1 and 2 have stopped at statement *enter* even though the *enter* action is always enabled. (Since there are a finite number of processes, it is always possible to choose a natural number that is greater than $num[p]$ for all processes p.) Hence, this behavior does not satisfy the assumption of weak fairness for all processes, so it is not a behavior of the algorithm.

TLC reports the violation because it isn't checking if the algorithm is starvation free; it's checking if a *model* of the algorithm is starvation free. In my model, *Nat* is defined to equal $0\,..\,5$. There is no element of $0\,..\,5$ greater than 5.

Hence, the *enter* action of processes 1 and 2 is not enabled if $num[3] = 5$. The model allows this behavior because weak fairness does not require a process to execute an action that is continually disabled.

TLC can check only models of the algorithm in which *Nat* is replaced with a bounded set of numbers, and any such model will allow behaviors that are not starvation free because some process p keeps setting $num[p]$ equal to the largest number in *Nat*. To check if the algorithm is starvation free, we have to tell TLC to ignore this class of behaviors. We do this by adding a *state constraint* to the model. A state constraint is a state predicate P (a formula containing no primes or temporal operators) that tells TLC to examine only behaviors in which each state satisfies P. More precisely, when it finds a reachable state s that does not satisfy P, it does not look for states that are reachable from s. (However, it will test if s satisfies the invariants it is checking.) For my model with *Nat* equal to $0 \,..\, 5$, I enter the state constraint

$$\forall p \in Procs : num[p] < 5$$

in the State Constraint field on the Advanced Options model page. TLC then reports that property *StarvationFree* is satisfied by the model.

The problem of errors that occur in a model but not in the actual algorithm can occur when checking any kind of property with TLC. However, it seems to be more common when checking liveness than when checking safety. For safety properties, a state constraint usually provides a satisfactory solution. For liveness properties, the constraint could easily cause TLC not to check actual behaviors of the algorithm that violate the property. For example, TLC could not find a failure of starvation freedom in the bakery algorithm caused by a process never entering its critical section because other processes kept setting their numbers to ever increasing values. We need to prove that this is impossible. Still, we should let TLC try to find whatever errors it can. There's no point trying to write a proof if TLC can find a counterexample.

Question 7.16 Find an algorithm that satisfies an invariant, but for which the invariant is violated by any model TLC can check—if the model doesn't use a state constraint. ANSWER

Let's now prove that the big-step bakery algorithm is starvation free. It, and the actual bakery algorithm, satisfy the stronger property of being first-come-first-served (FCFS). Not only does each process trying to enter its critical section do so, but it enters before any process that enters the bakery algorithm after it does. More precisely, if process p executes the *enter* statement before process q does, then p enters the critical section before q does. First-come-first-served is described formally by the property *FCFS*, defined as follows.

$$InNCS(p) \triangleq pc[p] = \text{“ncs”}$$
$$Waiting(p) \triangleq pc[p] = \text{“wait”}$$

$$FCFS \triangleq$$
$$\forall\, p,\, q \in Procs :$$
$$\Box(Waiting(p) \;\wedge\; InNCS(q) \;\wedge\; \Box\neg InCS(p) \;\Rightarrow\; \Box\neg InCS(q))$$

It is easy to see why this property holds. From $Waiting(p) \wedge \Box\neg InCS(p)$, we deduce that $Waiting(p)$ remains forever true, with $num[p] > 0$. From $InNCS(q)$ we can then deduce that if q tries to enter its critical section, it will set $num[q]$ greater than $num[p]$ and will wait forever in its **await** statement with $i = p$. Here is a more rigorous proof.

FCFS is a safety property$^{\Box}$; starvation freedom is a liveness property. A safety property cannot imply a liveness property—except in trivial cases. (For example, the safety property FALSE implies every liveness property.) Starvation freedom is implied by FCFS and deadlock freedom. The idea of the proof is simple. Suppose a process p is trying to enter its critical section. FCFS implies that no process that later tries to enter its critical section can succeed before p. Deadlock freedom implies that as long as p is trying to enter, processes must keep entering and leaving the critical section. Since there can be only a finite number of processes waiting at any time, eventually no process other than p will be able to enter the critical section, so it must do so. Here is a more rigorous proof of this result. Thus, to prove that the big-step bakery algorithm is starvation free, we just have to prove that it is deadlock free. This is left as an exercise:

Problem 7.17 Prove $Spec \Rightarrow DeadlockFree$.

7.8.2 Choosing the Grain of Atomicity

The big-step bakery algorithm isn't a useful algorithm because it assumes that process *self* can read $num[i]$ for all other processes i and set $num[self]$ all in a single step. The only way I know of making such an operation act like an atomic step is to put it in the critical section of a mutual exclusion algorithm—not helpful if we're trying to implement mutual exclusion. So, we want a finer-grained algorithm.

The algorithm's **process** declaration should look something like this, where the **with** statement of the Big-Step algorithm has been refined to a **while** loop that reads the values $num[i]$ individually, for all $i \neq self$, and then sets $num[self]$. I have not yet added the labels that specify exactly what the atomic steps are. We now consider how that should be done.

Dijkstra assumed that reading or writing a single word of memory is an atomic action. Viewing Dijkstra's paper from the perspective of our standard model, we would phrase this assumption as follows. The system's state consists of a collection of memory words that can be read and written by all the processes, together with a collection of registers, each local to a single process. An operation of a process may be taken to be atomic if it reads or writes at most one word of

memory. The operation may perform arbitrary operations on its local registers. We would justify this assumption by arguing that, because the operations to the local registers cannot affect or be affected by operations performed by another process, we can assume that they all occur at the same instant, which is the same instant as the read or write of memory (if there is such a read or write) is performed.

Memory and registers are meaningful concepts for a computer, but not for an algorithm. When computer scientists generalized from computers to processes, they (often implicitly) partitioned the state of an algorithm into elementary data items, some shared by multiple processes and others local to individual processes. Typically, an integer-valued variable was taken to be an elementary data item—tacitly assuming that an implementation would ensure that its value remained small enough to fit in a single word of physical memory. For now, let us take $num[i]$, $unchecked[i]$, $max[i]$, and $pc[i]$, for all processes i, to be the elementary data items of the bakery algorithm. (Remember that process i's copies of the process-local variables *unchecked* and *max* are the array elements $unchecked[i]$ and $max[i]$.)

Dijkstra's assumption translates to the requirement that an atomic action contain at most one read or write of at most one shared elementary data item. A closer look at its justification, which involves grouping together operations that are invisible to another process into a single atomic action, shows that we can weaken that requirement to the following:

> **Single Access Rule** A single atomic action of a process may contain either (a) a single write to a shared elementary data item or (b) a single read of a shared data item that may be written by another process (but not both).

For example, the following atomic action a

a: $num[self] := num[self] + max$;
  $unchecked[self] := Procs$;
b:

would satisfy the Single Access Rule for the bakery algorithm. It writes to the single data item $num[self]$, which is allowed by condition (a). It also reads $num[self]$ and $max[self]$, neither of which is written by any process other than *self*, and it writes $unchecked[self]$, which is not shared. (The constant *Procs* is not a data item because it is not part of the state.)

We want to represent the bakery algorithm as the coarsest-grain PlusCal algorithm possible (the one with the biggest atomic steps) that satisfies the Single Access Rule. Here is how we can add labels to do that. (Note that statements *e*2 and *wait* both contain two occurrences of the shared data item $num[i]$. We consider those two occurrences to represent a single read of $num[i]$, since a sensible implementation of those statements would read $num[i]$ only

once.) Before we examine this algorithm, let's take a more critical look at the Single Access Rule.

Let Π be an algorithm satisfying the rule. We can view it as an abstract model of a finer-grained algorithm $\widehat{\Pi}$ that more accurately models a real system. What we would like to be true is that the correctness of Π implies the correctness of $\widehat{\Pi}$, assuming that $\widehat{\Pi}$ maintains the atomicity of reads and writes to shared elementary data items. Whether or not it *is* true depends on what constitutes correctness. For example, suppose x is an array of shared data items and h is a process-local variable. A PlusCal algorithm Π containing the following action a

a: $h := h + 1$;
  $x[self] := h$;
b:

might satisfy the invariance property $\Box(x[i] = h[i])$ for every process i. However that property is not satisfied by the finer-grained version $\widehat{\Pi}$ containing the steps:

a: $h := h + 1$;
$a2$: $x[self] := h$;
b:

We shouldn't expect an invariant of Π to be an invariant of $\widehat{\Pi}$ if it depends on the values of process-local data items, since Π combines multiple operations to those items in a single atomic action. However, we are doing that when we apply the Single Access Rule to a mutual exclusion algorithm, whose correctness condition is the invariance of a formula depending on the values of the process-local data items $pc[i]$.

Even an invariant of Π depending only on shared data items need not be an invariant of $\widehat{\Pi}$. Suppose x is an array of shared data items, h is a process-local variable, and Π contains the statement:

a: **with** ($i \in Nat$) { $h := i$ } ;
  $x[self] := h$;
  await $h = 42$;
b:

(Examining its TLA$^+$ translation shows that statement a is a complicated way of writing $h := 42$; $x[self] := 42$.) Algorithm Π might satisfy the invariance property $\Box(x[i] = 42)$ for all processes i. Now let $\widehat{\Pi}$ be obtained by adding a label to this code:

a: **with** ($i \in Nat$) { $h := i$ } ;
  $x[self] := h$;
$a2$: **await** $h = 42$;
b:

(The modified code sets h and $x[self]$ to an arbitrary natural number and then stops unless $h = 42$.) Algorithm $\widehat{\Pi}$ does not satisfy that invariance property.

In this example, the invariant of Π fails to be an invariant of $\widehat{\Pi}$ because the section of code consisting of statements a and $a2$ of $\widehat{\Pi}$ that implements the atomic action a of Π allows executions that don't correspond to executions of that atomic action. It turns out that this is the only way there can be a safety property satisfied by Π that depends only on shared data items but is not satisfied by $\widehat{\Pi}$. In general, for any atomic action A of Π, let $\widehat{A}$ be the section of PlusCal code that implements A in $\widehat{\Pi}$. Any safety property satisfied by Π that depends only on shared data items is also satisfied by $\widehat{\Pi}$ if the following condition holds:

> For every atomic action A of Π and any states s and t, if executing $\widehat{A}$ starting in state s can produce state t, then executing A in state s can produce state t.

I will not try to state this result more rigorously. In most modern multiprocessor computers, each processor has its own cache, and reading or writing a single word of memory cannot be regarded as an atomic action. The rationale behind the Single Access Rule is therefore no longer valid. Understanding the Single Access Rule will help you determine the appropriate grain of atomicity for an algorithm. However, determining if an algorithm is a useful model of a real system requires an understanding of the algorithm, the system, and the properties you are trying to check.

I will let you convince yourself that the algorithm given above has an appropriate grain of atomicity for checking mutual exclusion, assuming that reads and writes of each data item $num[i]$ are atomic.

7.8.3 The Atomic Bakery Algorithm

We now develop the *atomic bakery algorithm*, so called because reads and writes of each $num[i]$ are taken to be atomic. I presented an initial attempt at such an algorithm above. Here is its complete ascii version. It does not implement mutual exclusion. Before reading any further, run TLC to see why not.

Here is the incorrect behavior that TLC produces for a model with $N \leftarrow 2$ and $Nat \leftarrow 0 \,..\, 4$.

- Both processes execute statement $e1$ and $e2$, reaching $e3$ with $max = 0$. Process 1 then executes $e3$, setting $num[1] = 2$, executes the *wait* loop (seeing $num[2] = 0$), and reaches cs.
- Process 2 then executes $e3$, setting $num[1] = 1$, executes the *wait* loop (seeing $num[1] = 2$, so $\langle num[2], 2\rangle \prec \langle num[1], 1\rangle$), and reaches cs, making $MutualExclusion$ false.

This incorrect execution would not occur if, instead of setting $num[self]$ to an arbitrary integer greater than max, it set $num[self]$ to $max + 1$. Make that change to the algorithm and use TLC to show that it is still incorrect.

With this change to the algorithm, TLC will find a behavior in which both processes reach $e3$ with $max = 1$. Process 2 then executes the waiting loop, sees $num[1] = 0$, and reaches cs. Process 1 then executes the waiting loop, sees $num[2] = 1$ and also reaches cs because $\langle 1, 1\rangle \prec \langle 1, 2\rangle$.

The scenario that must be prevented is processes p and q starting in their non-critical sections and:

- Process p reaches $e3$ after seeing $num[q] = 0$.
- Process q reaches its critical section before p sets $num[p]$.
- Process p then sets $num[p] \leq num[q]$, so p does not have to wait for process q before entering its critical section.

We prevent the second item in the scenario by requiring q to wait for p when q has left its non-critical section but not yet set $num[q]$. To do this, we have process q indicate that it is in this part of its code by setting $flag[q]$ to TRUE, where $flag$ is a global variable of the algorithm.

Here is the algorithm. The Single Access Rule requires that the body of the waiting loop consist of two separate actions: one reading $flag[i]$ and another reading $num[i]$, for each process i in $unchecked$. We therefore need an additional local variable, which we call nxt, to remember which process i has been chosen from $unchecked$. To simplify an invariance proof, we initialize the local variables to type-correct values even though those initial values are never used. ASCII version

You can let TLC check that this algorithm does satisfy mutual exclusion. On my computer, it checks a model with $N \leftarrow 3$ and $Nat \leftarrow 0\,..\,5$ in less than 1.5 minutes. How large a model do you have the patience to check? Note that checking this model also checks any model with $N \leq 3$ and N a subset of $0\,..\,5$, since it includes executions in which a smaller number of processes actually take steps and the $num[i]$ take only values in that subset. Writing a rigorous invariance proof of the atomic bakery algorithm is good practice.

Problem 7.18 Write a rigorous proof that the atomic bakery algorithm's specification implies $\Box MutualExclusion$. (Use TLC to help you find the inductive invariant.) ANSWER

The bakery algorithm has the inelegant property that the values of the data items $num[i]$ can get arbitrarily large. We can keep them from getting too large by replacing statement $e3$ with $num[self] := max + 1$. It's easy to see that this keeps $num[i]$ at most equal to the number of times some process has tried to enter its critical section. So, if processes enter their critical section no more than

once a nanosecond, the value of $num[i]$ will fit in 64 bits of memory for 50000 years.

It seems that by restricting $e3$ in this way, the non-zero values of $num[i]$ at any one time will all lie within about N of one another, so we could allow the values of the $num[i]$ to cycle through a finite set of integers. In the following problem, you will show that this is not possible.

Problem 7.19 If $N \geq 3$, then even if $e3$ is replaced by $num[self] := max + 1$, for any two natural numbers M and P, there is an execution in which $num[i] = M$ and $num[j] = P$ for two processes i and j. HINT

However, there is a two-process algorithm with a bounded set of values.

Problem 7.20 For $N = 2$, find a version of the bakery algorithm in which the there is a finite set of values that always contains the value of each $num[i]$. ANSWER

For liveness, as usual we turn the **process** declaration into a **fair process** and allow a process to stop in its non-critical section by putting "`-`" after the label "`ncs:`". The proof that the atomic bakery algorithm is starvation free is essentially the same as for the big-step algorithm.

7.8.4 The Real Bakery Algorithm

While it may be useful, an algorithm that assumes atomic reads and writes of shared data items cannot really be said to solve the mutual exclusion problem. Implementing those atomic reads and writes would seem to require mutually exclusive access to the data items. From a scientific point of view, an algorithm that assumes lower-level mutual exclusion cannot be said to solve the mutual exclusion problem.

The most remarkable property of the bakery algorithm is that it does not require atomic reads and writes of individual data items. It needs only two properties of reads and writes: (i) a read that does not overlap a write obtains the correct value, and (ii) any read obtains a value of the correct type. For example, a read of $num[i]$ by a process other than process i returns the current value of $num[i]$ if i is not writing the value during that read, and it always returns a natural number. Thus, $num[i]$ could be implemented with an array of bits; process i can write $num[i]$ by writing the bits one at a time in any order; another process can read $num[i]$ by reading the bits in any order. Reading and writing of an individual bit need not even be atomic.

Here is one way to model such non-atomic reads and writes of $num[i]$: Process i sets $num[i]$ to a number m by first setting $num[i]$ to a special value $\perp$ and then setting it to m. A read by another process reads the current value of $num[i]$ atomically, and it returns an arbitrarily chosen natural number if it sees $num[i]$ equal to $\perp$.

This way of modeling reads and writes is probably the best one for model checking, since it seems to minimize the number of reachable states. However, there is another way that is more convenient for reasoning about the algorithm: A process sets $num[i]$ to m by first performing a sequence of writes of arbitrarily chosen natural numbers to $num[i]$, and then setting $num[i]$ to m. A read just atomically reads the current value of $num[i]$. Here's the PlusCal code for the bakery algorithm's *exit* statement that sets $num[self]$ to 0:

$$\begin{array}{lll} exit: & \textbf{either } \{ & \textbf{with } (\ k \in Nat\ ) \ \{\ num[self] := k\ \}\ ; \\ & & \quad \textbf{goto } exit\ \} \\ & \textbf{or} & \{\ num[self] := 0\ \} \end{array}$$

Here is the algorithm. (You can compare it to the atomic bakery algorithm.) Note that because $flag[self]$ is Boolean-valued, we can simplify our modeling of writes to it. Here is the ASCII version.

TLC can check this algorithm on a model with $N \leftarrow 2$ and $Nat \leftarrow 0\,..\,6$ in a couple of seconds. It takes a couple of minutes for a model with $N \leftarrow 3$ and $Nat \leftarrow 0\,..\,3$. TLC will not check a large enough model to give us very much confidence in the algorithm's correctness; that requires a proof.

Problem 7.21 Write a rigorous invariance proof that the specification *Spec* of the bakery algorithm implies $\Box MutualExclusion$. HINT

Making the **process** declaration a **fair process** does not ensure that any process ever enters its critical section. For example, a process could loop forever in step $e1$, always choosing to execute the **either** clause. We need an additional fairness condition to require that, if a process keeps executing $e1$, then it will eventually execute the statement's **or** clause. I think it is impossible to express this requirement in PlusCal. However, it is expressed in TLA$^+$ by weak fairness of the action $e1(i) \land (pc'[i] \neq pc[i])$, for each process i. When process i is looping at $e1$, this action is continuously enabled. Weak fairness implies that this action must eventually occur, which means that the $e1$ **or** clause must eventually be executed. Similar fairness requirements are needed to prevent infinite looping at $e3$, $e4$, and *exit*. We can express these requirements with the formula

This action is equivalent to $e1(i) \land (pc'[i] = \text{“}e2\text{”})$.

$$\begin{array}{l} \forall\, i \in Procs : \mathrm{WF}_{vars}(\ \land\ e1(i) \lor\ \ e3(i) \lor e4(i) \lor exit(i) \\ \qquad\qquad\qquad\qquad\quad \land\ (pc'[i] \neq pc[i])) \end{array}$$

Define *FairSpec* to be the conjunction of the formula *Spec* of the algorithm's TLA$^+$ translation with this formula. Use TLC to check that algorithm *FairSpec* is deadlock free and starvation free. (Remember that you have to use a state constraint when checking starvation freedom.) Checking liveness properties takes TLC longer that checking safety properties, so you may want to use smaller models than you used to check mutual exclusion.

The proofs of these liveness properties have the same general form as the proofs for the other versions of the bakery algorithm

8 The Bounded Channel and Bounded Buffer

8.1 The Bounded Channel

We now consider a two-process system in which a sender process puts messages in a *channel* that are removed in sequence by a receiver process. This is a FIFO (first in, first out) channel, meaning that messages are received in the order in which they are sent. We might draw a picture of the system like this:

Such pictures convey almost no useful information, but they help many people understand an actual specification.

We consider a *bounded* channel, which is one that can hold at most some number N of messages. If the channel contains N messages, then the sender cannot send another message until a message is removed from the channel by the receiver.

The specification uses a single variable *ch* that describes the state of the channel. The value of *ch* is the sequence of messages that have been sent but not received. In TLA$^+$, a finite sequence is the same as a tuple. Tuples are explained in Section 15.1 and Section 15.5. However, for now, you will need to know only the following operators that are defined in the standard *Sequences* module.

$Seq(S)$	The set of all finite sequences with elements in S. For example, $\langle 0, 42\rangle$ is an element of $Seq(Nat)$.
$Len(s)$	The length of the finite sequence s. For example: $Len(\langle 0, 42\rangle) = 2 \quad \text{and} \quad Len(\langle\,\rangle) = 0$
$Append(s, e)$	The sequence obtained by appending the element e to the end (as the right-most element) of the finite sequence s. For example: $Append(\langle 4, \text{“abc”}, -2\rangle, \text{“d”}) = \langle 4, \text{“abc”}, -2, \text{“d”}\,\rangle$
$Tail(s)$	The sequence obtained from the nonempty finite sequence s by deleting its first element. For example: $Tail(\langle 4, \text{“abc”}, -2\rangle) = \langle\, \text{“abc”}, -2\rangle$

With these operators, writing the TLA$^+$ specification is straightforward. In the Toolbox, create a new spec named *BoundedChannel*. The module begins as follows, where the constant *Msg* is the set of all possible messages that can be sent.

EXTENDS *Integers*, *Sequences*

```
EXTENDS Integers, Sequences
```

CONSTANT *Msg*, *N*

```
CONSTANT Msg, N
```

ASSUME $N \in Nat \setminus \{0\}$

```
ASSUME N \in Nat \ {0}
```

VARIABLE *ch*

```
VARIABLE ch
```

Since the value of *ch* is always the sequence of messages that have been sent so far, which is initially the empty sequence, the type-correctness invariant and the initial predicate should be:

$TypeOK \triangleq ch \in Seq(Msg)$

```
TypeOK == ch \in Seq(Msg)
```

$Init \triangleq ch = \langle\rangle$

```
Init == ch = <<>>
```

The system can take two possible kinds of steps: one that sends a message and one that receives a message. These are described by the two actions that we call *Send* and *Rcv*. A *Send* step can append any message to *ch*. The relation between the new and old values of *ch* is described by the formula

$$ch' = Append(ch, v)$$

for some nondeterministically chosen message v. In other words, there must exist some message v such that this formula holds. Thus the change to *ch* is described by

$$\exists\, v \in Msg : ch' = Append(ch, v)$$

Observe how the nondeterministic choice of v is expressed by existential quantification. In general, nondeterminism of a system is represented mathematically by existential quantification in the next-state action. It's important that you understand why this is true, so you should understand why this formula describes a "nondeterministic assignment to *ch*".

A *Send* step can be performed only when there are fewer than N messages in the channel—that is, only when the length of the sequence *ch* is less than N. Thus, we can define *Send* to equal

$$\begin{array}{l} \land\ Len(ch) < N \\ \land\ \exists\, v \in Msg : ch' = Append(ch, v) \end{array}$$

However, I prefer the following definition:

$$\begin{array}{lll} Send & \triangleq & \land\ Len(ch) \neq N \\ & & \land\ \exists v \in Msg : ch' = Append(ch, v) \end{array}$$

```
Send == /\ Len(ch) # N
        /\ \E v \in Msg : ch' = Append(ch, v)
```

Since $Len(ch)$ is always at most N, these two actions define equivalent specifications.

Receiving a message is represented by an action *Rcv* that removes the first element from *ch*. This can be done only when *ch* has at least one element. We can therefore define *Rcv* as follows:

$Rcv \triangleq \land Len(ch) \neq 0$
$\quad\quad\land ch' = Tail(ch)$

```
Rcv == /\ Len(ch) # 0
       /\ ch' = Tail(ch)
```

A step of the system is either a *Send* step or a *Rcv* step, so the next-state action is defined by:

$Next \triangleq Send \lor Rcv$

```
Next == Send \/ Rcv
```

The complete specification *Spec* is then

$Spec \triangleq Init \land \Box[Next]_{\langle ch \rangle}$

```
Spec == Init /\ [][Next]_<<ch>>
```

Save the module, which should look like this.

Let's use TLC to look for errors by checking that *TypeOK* is an invariant. Create a new model. Assign the value 4 to N, and assign to *Msg* a set consisting of the three model values m_1, m_2, and m_3 by selecting the *Set of model values* option, typing the value

```
{m1, m2, m3}
```

and finishing by choosing the *Leave untyped* option.

The specification does not distinguish between the elements of *Msg*. This means that given any behavior that satisfies our specification, permuting the values of *Msg* the same way in all the states yields another behavior that satisfies the specification. In this case, we say that the specification is *symmetric under permutations of the set Msg*. Knowing that a specification is symmetric under permutations of a set allows TLC to reduce the number of states that it must explore. We can tell TLC that our specification is symmetric under permutations of *Msgs* by checking the *Symmetry set* option when we assign the set of model values to it. Edit the substitution for *Msgs* to do that and observe the number of distinct states that TLC finds. You will see that it finds approximately $1/6^{\text{th}}$ as many states as it did before, because there are 6 different permutations of the 3-element set *Msgs*.

When you tell TLC that a set is a symmetry set of the specification, TLC doesn't check that it really is. If it's not, this could cause TLC to miss an error by not checking all the states it should.

TLC can make use of symmetry only under a set of model values. It could not take advantage of symmetry if we had assigned a set of numbers to *Msg*.

Question 8.1 What is the meaning of the bounded channel specification if *Msg* is the empty set? How does TLC confirm your answer? ANSWER

Our specification may not seem very interesting, since the sender sends arbitrarily chosen messages that the receiver simply throws away. The Bounded Channel specification describes just the part of a system used to send messages from one process to another. A complete specification of the system would describe how the messages are created and what the receiver does with the ones it receives.

8.2 The Bounded Channel in PlusCal

8.2.1 Getting Started

We now rewrite the Bounded Channel specification in PlusCal. Create a new module named *PCalBoundedChannel*. It begins like the *BoundedChannel* module:

EXTENDS *Integers*, *Sequences*	`EXTENDS␣Integers,␣Sequences`
CONSTANT *Msg*, *N*	`CONSTANT␣Msg,␣N`
ASSUME $N \in Nat \setminus \{0\}$	`ASSUME␣N␣\in␣Nat␣\␣{0}`

Instead of writing the initial predicate and next-state action in TLA$^+$, we write PlusCal code that produces them. As before, the algorithm goes in a multi-line comment. It begins with the **--algorithm** keyword and its name, which we take to be *BChan*; and it next declares the variable *ch* with its initial value:

```
(***************************************************
--algorithm BChan {
   variable ch = ⟨⟩;

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

Instead of having a simple body like the one-bit clock and Euclid's algorithm, the body of the *BChan* algorithm consists of two processes, which we call *Send* and *Rcv*:

```
process (Send = "S")
 { \* process body
 }

process (Rcv = "R")
 { \* process body
 }
```

The first **process** statement declares the process's name to be *Send* and declares it to have the string "S" as its *identifier*. Similarly, the second **process** statement declares a process named *Rcv* with identifier "R".

A process's identifier is usually a simple value like an integer or a string, but it could be any value—for example, the pair $\langle$ "proc", 42$\rangle$. However, all processes in an algorithm must have identifiers that are unequal to one another. This means that we cannot let *Send* have identifier "S" and *Rcv* have identifier $\langle$ "Rcv" $\rangle$, because the semantics of TLA$^+$ does not determine whether or not the string "S" equals the one-tuple $\langle$ "Rcv" $\rangle$. In practice, this means that we usually let all the process identifiers of an algorithm have the same "type"—for example, all strings or all numbers.

The body of each process is repeatedly executed forever, so it consists of a **while** (TRUE) loop. For the *Send* process, the loop is

```
while (TRUE)  { await Len(ch) ≠ N ;
                with (v ∈ Msg) { ch := Append(ch, v) }
              }
```

The **await** statement can be executed only when its predicate, $Len(ch) \neq N$, equals TRUE, in which case the next statement can then be executed. The **with** statement nondeterministically chooses an arbitrary value v in the set Msg and executes its body, in this case the assignment statement, for that value of v. The body of process Rcv is similar, except ch is set to the tail of its current value.

```
while (TRUE)  { await Len(ch) ≠ 0 ;
                ch := Tail(ch)
              }
```

Save the file, and run the translator. It reports something like this:

```
-- Missing labels at the following locations:  line 12, column
8 line 18, column 9
```

where the line and column locations point to the two **while** statements. Let's now see what the problem is.

8.2.2 Specifying the Grain of Atomicity

To understand why the translator is complaining, let's examine the PlusCal version of the one-bit clock specification. The algorithm's body consists of the loop:

```
while (TRUE) { if (b = 0) b := 1 else b := 0 }
```

Most people think of an execution of one iteration of the **while** loop as consisting of the following three steps:

1. Evaluate the **while** statement's test and find that it is true.
2. Evaluate the test $b = 0$ of the **if** statement.
3. (a) If the **if** statement's test equals TRUE, then execute the assignment $b := 1$, else
   (b) If the **if** statement's test equals FALSE, then execute the assignment $b := 0$.

However, the TLA$^+$ translation of this body yields a next-state relation in which each iteration of the **while** loop is executed as a single step (state change). A single step of a behavior is often called an *atomic action*, and choosing what constitutes a single step is often called the choosing the *grain of atomicity*. Why did the translator choose this grain of atomicity?

For a sequential algorithm (one with no processes), the translator by default chooses the coarsest grain of atomicity (the fewest steps) that it can. The translator cannot combine two iterations of a **while** loop into a single step, so the best it can do is to make each iteration of the loop a single step. We usually don't care about the grain of atomicity of a sequential algorithm, so we prefer a coarse grain of atomicity because it makes model checking more efficient (there are fewer reachable states) and it makes proving correctness of the algorithm simpler (because the next-state action is simpler).

The grain of atomicity is crucial to the correctness of a multiprocess (concurrent) algorithm. For example, consider the following algorithm:

```
--algorithm Increment { variable x = 0;
                        process (A = 1) { x := x + 1 }
                        process (B = 2) { x := x + 1 }
                      }
```

The algorithm starts with x equal to 0, and each process simply executes the assignment statement $x := x + 1$ and halts. If both assignments are executed as a single (atomic) step, then the algorithm terminates (when both processes terminate) with x equal to 2. However, if the assignment statement is executed in two steps, the first step evaluating $x + 1$ and the second step assigning the value to x, then the algorithm can terminate with x equal to either 1 or 2.

Question 8.2 (a) When execution of each assignment statement takes two steps, describe executions that end with x equal to 1 and with x equal to 2. ANSWER

(b) If the assignment statements are executed in more than two steps, what other possible values can x have upon termination?

By default, the translator requires the user to specify the grain of atomicity for a multiprocess algorithm. This is done by adding labels to the algorithm. A single step consists of an execution from one label to the next. There are certain rules about where labels must or must not appear. The most important ones are:

- The first statement of a process must have a label.
- Each **while** statement must be labeled.

The translator will tell you if you violate any of the labeling rules. In the unlikely event that you can't figure out from the error message what you need to do to fix a labeling problem, see Section 2.7 of the PlusCal language manual. The language manual also explains how to tell the translator to insert any necessary labels that are missing.

For algorithm *BChan*, we can satisfy both of the rules above by labeling the **while** statement in each process. Since a step consists of execution from one label to the next, using only these labels means that an entire execution

of one loop iteration is a single step. This is the same grain of atomicity as in the TLA$^+$ specification of module *BoundedChannel*, so let's use it. A label can consist of any sequence of letters, digits, and underscore (_) characters containing at least one letter. Let's use the labels s and r, to get this complete algorithm:

```
--algorithm BChan {
  variable ch = ⟨⟩;
  process (Send = "S")
    { s: while (TRUE)
           { await Len(ch) ≠ N ;
             with (v ∈ Msg) { ch := Append(ch, v) }
           }
    }
  process (Rcv = "R")
    { r: while (TRUE)
          {await Len(ch) ≠ 0 ;
           ch := Tail(ch)
          }
    }
}
```

The ASCII version of the module with the algorithm is here. Run the translator (from the File menu or by typing control+t). The TLA$^+$ translation is the same as the specification in module *BoundedChannel* except that:

- The formatting is a little different.
- The translator adds a definition of *ProcSet*, which it is not clever enough to realize is not needed for this algorithm.
- The translator adds the definitions of *vars* and *Spec*, which we will discuss later.

The sender's statement

await $Len(ch) \neq N$

is translated to the conjunct $Len(ch) \neq N$ of the process's action. A conjunct of an action that contains no primes is an enabling condition for the action. Observe how the **with** statement is translated to an existential quantification. As we observed above, nondeterministic choice is expressed in an action by existential quantification.

8.3 The Bounded Buffer

We next give a "lower-level" PlusCal implementation of the bounded channel. But first, we need a brief mathematical digression.

8.3.1 Modular Arithmetic

The *Integers* module defines the *modulus* operator $\%$ so that $a \,\%\, b$ is the remainder when a is divided by b. More precisely, for any integers a and b with b positive, $a \,\%\, b$ is the unique number satisfying the two conditions:

$$a \,\%\, b \in 0\,..\,(b-1) \qquad \exists\, q \in \mathit{Int} \,:\, a = b * q + a \,\%\, b$$

For any positive integer K, let us define the operators $+_K$ and $-_K$ by

$$a +_K b \;\triangleq\; (a + b) \,\%\, K$$
$$a -_K b \;\triangleq\; (a - b) \,\%\, K$$

The symbols $+_K$ and $-_K$ can't be written in TLA$^+$, but it will be convenient to use them anyway. We are interested in these two operators when applied to numbers in the set $0\,..\,(K-1)$. To understand their meaning, we write this set of numbers in a circle, as shown in this picture.

If a and b are in $0\,..\,(K-1)$, then $a +_K b$ is the number obtained by starting at a and moving clockwise b numbers. For example, we see from the picture that $(K-2) +_K 5$ equals 3. We can characterize $a -_K b$ as the distance from b to a going clockwise around the circle. For example, $3 -_K (K-2)$ equals 5.

If you have studied group theory, you may recognize $0\,..\,(K-1)$ as the Abelian (commutative) group known to mathematicians as Z_K, where $+_K$ and $-_K$ are its addition and subtraction operators. This means that, when applied to elements of $0\,..\,(K-1)$, the operators $+_K$ and $-_K$ obey most of the same rules of arithmetic that $+$ and $-$ do on the set *Int* of integers. For example, if a, b, and c are in $0\,..\,(K-1)$, then:

$$a +_K (b -_K c) \;=\; (a -_K c) +_K b$$

8.3.2 The Bounded Buffer Algorithm

Our *bounded buffer* algorithm implements the bounded channel by implementing the sequence *ch* of messages with an array *buf*. Each message in *ch* is contained in some element $\mathit{buf}[i]$ of *buf*. Since *ch* can contain up to N messages, *buf* must contain at least N elements. We let it be an N-element array indexed by $0\,..\,(N-1)$.

What programmers call *arrays* are a special class of what mathematicians call *functions*. The *index set* of an array is called by mathematicians the *domain* of the function. Programmers think of an array A as a collection of "containers", one for each element in its index set; they think of $A[i]$ as the contents of the i^{th}container of A. Mathematicians think of a function A as a rule that assigns to each element i in its domain a value $A(i)$.

Since a PlusCal "array" can be any mathematical function, not just the special kinds of functions allowed by programming languages, we use the terminology of mathematicians rather than programmers. However, TLA$^+$ maintains

Any function?

the programmer's notation of writing $A[i]$ instead of $A(i)$ for the value assigned by A to the element i of its domain.

For sets D and R, mathematicians define the *set of functions from D to R* to be the set of all functions f with domain D such that $f[x]$ is an element of R for all x in D. This set is written in TLA$^+$ as $[D \to R]$.

In our bounded buffer algorithm, the variable buf is a function from $0\,..\,(N-1)$ to Msg, the set of all messages. More precisely, the following formula (which describes the "type" of the variable buf) will be an invariant of the algorithm:

$$buf \in [0\,..\,(N-1) \to Msg]$$

In addition to buf, our algorithm uses two variables p and c whose values are "pointers" to elements in $0\,..\,(N-1)$. The value of c points to the buffer element that contains (or will contain) the next message to be received; the value of p points to the buffer element into which the next message sent will be put.

As explained in Section 8.3.1 above, we think of the elements of the domain $0\,..\,(N-1)$ of buf as being arranged in a ring. The sequence of messages that have been sent but not yet received is the sequence of elements in the buffer starting from one pointed to by c and, moving clockwise, ending with the element right before the one pointed to by p. In this picture, the sequence of messages equals

$$\langle\, buf[N-2],\ buf[N-1],\ buf[0],\ buf[1],\ buf[2] \,\rangle$$

Observe that the length of this sequence is 5, which equals $3 -_N (N-2)$, where $-_N$ is defined in Section 8.3.1. As this example illustrates, in general the length of the sequence of messages is $p -_N c$.

However, this can't be correct. The value of $p -_N c$ is an integer from 0 through $N-1$, but the bounded channel can contain sequences of messages of length N. The problem is that if c points to the buffer element containing the next message to be received and p points to the element into which the next message sent is to be put, then p equals c both when the channel is empty (equals the empty sequence) and when it is full (it is a sequence of N messages). We must find a way to disambiguate these two cases.

One solution is to use an $N+1$ element buffer, with one buffer element always unused. The channel is then empty when p and c point to the same buffer element; it is full when p points to the element just before the one pointed to by c.

Instead, we use what I find to be a more elegant solution. We let p and c be elements of $0\,..\,(2N-1)$, and we let $p \,\%\, N$ and $c \,\%\, N$ be the buffer pointers—as shown in this picture. The length of the sequence of messages is then equal to $p -_{2N} c$. The buffer is empty when p equals c and is full when p equals $c +_{2N} N$. In general, the sequence of messages is

$$\langle\, buf[c \,\%\, N],\ buf[(c +_{2N} 1) \,\%\, N],\ \ldots,\ buf[(p -_{2N} 1) \,\%\, N] \,\rangle$$

Instead of $2N$, we could use kN for any integer $k > 1$; for simplicity, we use $2N$.

We can now write our algorithm. Open the Toolbox and, in the same folder (directory) as module *PCalBoundedChannel*, create a new module *PCalBoundedBuffer*. The module begins just like module *PCalBoundedChannel*, except we will not need to extend the *Sequences* module.

EXTENDS *Integers*	`EXTENDS Integers`
CONSTANT *Msg*, *N*	`CONSTANT Msg, N`
ASSUME $N \in Nat \setminus \{0\}$	`ASSUME N \in Nat \ {0}`

We next define $\oplus$ and $\ominus$ (typed `(+)` and `(-)`) to be the operators that we have been calling $+_{2N}$ and $-_{2N}$.

$a \oplus b \triangleq (a + b) \,\%\, 2 * N$	`a (+) b == (a + b) % 2*N`
$a \ominus b \triangleq (a - b) \,\%\, 2 * N$	`a (-) b == (a - b) % 2*N`

We now come to the algorithm, which we name *BBuf*. As usual, it appears in a comment in the module. Click here to see the algorithm's code.

The **variables** statement declares the three variables *buf*, *p*, and *c* along with their initial values. We let *buf* initially equal an arbitrary function from $0\,..\,(N-1)$ to *Msg*. Since we want the sequence of messages in the channel to be initially empty, we must let *p* and *c* both initially equal the same element of $0\,..\,(N-1)$. For simplicity, we let them both equal 0.

Question 8.3 How would you write the declarations of *p* and *c* so they both initially equal the same arbitrary element of $0\,..\,(N-1)$? ANSWER

In this type of bounded buffer algorithm, it is customary to call the two processes the *producer* and *consumer*, rather than the *sender* and *receiver*. We therefore name the processes *Producer* and *Consumer*, giving them the identifiers "P" and "C", respectively. As in the bounded channel's PlusCal code, each process executes a **while** (TRUE) loop whose body is a single atomic action. I used the labels *p*1 and *c*1 because later we will add labels *p*2, *p*3, and *c*3. You should have no trouble understanding these processes by comparing them with the processes of the bounded channel algorithm. Like the receiver of our bounded channel algorithm, the consumer process does nothing with the messages that the producer puts into the buffer. Note

Here is the ASCII version of the *PCalBoundedBuffer* module. Use it to create a new specification in the Toolbox.

8.3.3 The TLA$^+$ Translation

Run the PlusCal translator (from the File menu or by typing control+t) and examine the TLA$^+$ translation. The definitions of the initial predicate *Init* and next-state action *Next* are quite similar to the ones for the bounded channel algorithm's PlusCal code. They should be easy to understand, except for the second conjunction of the *Producer* action, which is the translation of the PlusCal statement:

with $(v \in Msg)$ { $buf[p \,\%\, N] := v$ }

Most readers probably expect this to be translated as:

$$\exists v \in Msg : buf[p \,\%\, N]' = v$$

While this formula specifies the new value of $buf[p \,\%\, N]$, it doesn't specify the new value of $buf[i]$ for any i other than $p \,\%\, N$. In fact, it doesn't even assert that the new value of buf is a function with domain $0\,..\,(N-1)$. For example, the formula can be satisfied by letting buf' be a function whose domain is the singleton set $\{p \,\%\, N\}$. The correct translation of the PlusCal statement is a formula

$$\exists v \in Msg : buf' = f$$

where f is a function with the same domain as buf, such that $f[p \,\%\, N] = v$ and $f[i] = buf[i]$ for all other values i in the domain of buf. That function f is written in TLA$^+$ as:

$$[buf \text{ EXCEPT } ![p \,\%\, N] = v]$$

What do the EXCEPT and the ! mean?

We continue to ignore the definition of *Spec* for a little while longer.

You should now use TLC to check that you haven't made any mistakes. Add the following type-correctness invariant to the module, after the TLA$^+$ translation.

$$\begin{array}{lll} TypeOK \triangleq & \wedge\, buf \in [0\,..\,(N-1) \to Msg] \\ & \wedge\, p \in 0\,..\,(2*N-1) \\ & \wedge\, c \in 0\,..\,(2*N-1) \end{array}$$

```
TypeOK ==  /\ buf \in [0..(N-1) -> Msg]
           /\ p \in 0..(2*N-1)
           /\ c \in 0..(2*N-1)
```

Create a small model that checks if *TypeOK* is an invariant, and run TLC on it. TLC should find no error.

8.4 The Bounded Buffer Implements the Bounded Channel

8.4.1 Refinement Mappings

I introduced the bounded buffer algorithm as an implementation of the bounded channel. I will now explain exactly what that means. Recall that the sequence of messages in the channel is represented by the sequence of elements

$$\langle\, buf[c \,\%\, N],\ buf[(c \oplus 1) \,\%\, N],\ \ldots,\ buf[(p \ominus 1) \,\%\, N] \,\rangle$$

Let us call this sequence of messages $\overline{ch}$. For example, in this picture, $\overline{ch}$ equals

$$\langle\, buf[N-2],\ buf[N-1],\ buf[0],\ buf[1],\ buf[2] \,\rangle$$

Now suppose $N = 3$ and the set Msg of messages consists of the set of all one-character strings. Consider the following possible behavior of the algorithm,

which I will call σ. The behavior σ begins in an initial state with $buf[0] =$ "a", $buf[1] =$ "b", $buf[2] =$ "c", and continues by having the producer send the message "u", then send the message "v", and the receiver then receive the first message. (I have used an *ad hoc* method for writing the value of the function buf; its meaning should be clear.)

$$\left[\begin{array}{ll} p & = 0 \\ c & = 0 \\ buf & = 0 :> \text{"a"} \\ & \quad 1 :> \text{"b"} \\ & \quad 2 :> \text{"c"} \end{array}\right] \rightarrow \left[\begin{array}{ll} p & = 1 \\ c & = 0 \\ buf & = 0 :> \text{"u"} \\ & \quad 1 :> \text{"b"} \\ & \quad 2 :> \text{"c"} \end{array}\right] \rightarrow \left[\begin{array}{ll} p & = 2 \\ c & = 0 \\ buf & = 0 :> \text{"u"} \\ & \quad 1 :> \text{"v"} \\ & \quad 2 :> \text{"c"} \end{array}\right] \rightarrow \left[\begin{array}{ll} p & = 2 \\ c & = 1 \\ buf & = 0 :> \text{"u"} \\ & \quad 1 :> \text{"v"} \\ & \quad 2 :> \text{"c"} \end{array}\right] \rightarrow \cdots$$

Below each state in behavior σ, we now write the value of $\overline{ch}$ in that state.

$$\left[\begin{array}{ll} p & = 0 \\ c & = 0 \\ buf & = 0 :> \text{"a"} \\ & \quad 1 :> \text{"b"} \\ & \quad 2 :> \text{"c"} \end{array}\right] \rightarrow \left[\begin{array}{ll} p & = 1 \\ c & = 0 \\ buf & = 0 :> \text{"u"} \\ & \quad 1 :> \text{"b"} \\ & \quad 2 :> \text{"c"} \end{array}\right] \rightarrow \left[\begin{array}{ll} p & = 2 \\ c & = 0 \\ buf & = 0 :> \text{"u"} \\ & \quad 1 :> \text{"v"} \\ & \quad 2 :> \text{"c"} \end{array}\right] \rightarrow \left[\begin{array}{ll} p & = 2 \\ c & = 1 \\ buf & = 0 :> \text{"u"} \\ & \quad 1 :> \text{"v"} \\ & \quad 2 :> \text{"c"} \end{array}\right] \rightarrow \cdots$$

$\overline{ch} = \langle\rangle$ $\quad$ $\overline{ch} = \langle \text{"u"} \rangle$ $\quad$ $\overline{ch} = \langle \text{"u"}, \text{"v"} \rangle$ $\quad$ $\overline{ch} = \langle \text{"v"} \rangle$

We now erase the state and just keep the sequence of values of $\overline{ch}$.

$$[\,\overline{ch} = \langle\rangle\,] \;\rightarrow\; [\,\overline{ch} = \langle \text{"u"} \rangle\,] \;\rightarrow\; [\,\overline{ch} = \langle \text{"u"}, \text{"v"} \rangle\,] \;\rightarrow\; [\,\overline{ch} = \langle \text{"v"} \rangle\,] \;\rightarrow\; \cdots$$

Finally, we erase the overbars (replacing $\overline{ch}$ by ch), to get the following sequence that I will call $\overline{\sigma}$.

$$[\,ch = \langle\rangle\,] \;\rightarrow\; [\,ch = \langle \text{"u"} \rangle\,] \;\rightarrow\; [\,ch = \langle \text{"u"}, \text{"v"} \rangle\,] \;\rightarrow\; [\,ch = \langle \text{"v"} \rangle\,] \;\rightarrow\; \cdots$$

We say that the bounded buffer algorithm *implements* the bounded channel *under the refinement mapping* $ch \leftarrow \overline{ch}$ iff for every possible behavior σ of the bounded buffer specification, the sequence $\overline{\sigma}$ constructed in this way is a possible behavior of the bounded channel specification. We say that σ is *mapped to* $\overline{\sigma}$ by this refinement mapping.

Here's what we have just done. Let a *channel state* be an assignment of a value to the variable ch, and let a *buffer state* be an assignment of values to the variables p, c, and buf. We informally defined $\overline{ch}$ to be a mapping from buffer states to values—for example, it assigns to the buffer state

$$\left[\begin{array}{ll} p & = 2 \\ c & = 1 \\ buf & = 0 :> \text{"u"} \\ & \quad 1 :> \text{"v"} \\ & \quad 2 :> \text{"c"} \end{array}\right]$$

the value $\langle$ "v" $\rangle$. This defines a mapping from buffer states to channel states, which maps any buffer state s to the channel state $\overline{s}$ that assigns the value of $\overline{ch}$ in state s to the variable ch. This in turn defines a mapping from any sequence

$$s_1 \to s_2 \to s_3 \to \cdots$$

of buffer states to the sequence

$$\overline{s_1} \to \overline{s_2} \to \overline{s_3} \to \cdots$$

of channel states. For any sequence σ of buffer states, we let $\overline{\sigma}$ be the corresponding sequence of channel states. We say that the bounded buffer algorithm implements the bounded channel under the refinement mapping $ch \leftarrow \overline{ch}$ iff, for every behavior allowed by the bounded buffer algorithm's specification, the behavior $\overline{\sigma}$ is allowed by the bounded channel's specification.

Let $Spec_C$ be the specification *Spec* of the bounded channel, defined in module *BoundedChannel* or module *PCalBoundedChannel* (the two formulas are equivalent). Define $\overline{Spec_C}$ to be the formula obtained from $Spec_C$ by substituting the expression $\overline{ch}$ for the variable ch. Let's now see what $\overline{Spec_C}$ means.

We first consider the part of $Spec_C$ asserting the bounded channel's initial predicate $ch = \langle\rangle$. Formula $Spec_C$ equals $(ch = \langle\rangle) \wedge \ldots$, so $\overline{Spec_C}$ equals $(\overline{ch} = \langle\rangle) \wedge \overline{\ldots}$, where $\overline{\ldots}$ is obtained from $\ldots$ by substituting $\overline{ch}$ for ch. The state $\overline{s_1}$ assigns to ch the value that state s_1 assigns to $\overline{ch}$, so $\overline{ch} = \langle\rangle$ is true of state s_1 iff $ch = \langle\rangle$ is true of the state $\overline{s_1}$. Hence $ch = \langle\rangle$ (viewed as a temporal formula) is true of the behavior $\overline{\sigma}$ iff $\overline{ch} = \langle\rangle$ is true of the behavior σ.

The same reasoning shows that any subexpression e of $Spec_C$ is true of $\overline{\sigma}$ iff the corresponding subexpression $\overline{e}$ of $\overline{Spec_C}$ is true of σ. In particular, the entire specification $Spec_C$ is true of $\overline{\sigma}$ iff $\overline{Spec_C}$ is true of σ.

Let $Spec_B$ be the formula *Spec* of *PCalBoundedBuffer*, which is the specification of the bounded buffer. We can now reason as follows:

The bounded buffer implements the bounded channel

iff for every behavior σ of the bounded buffer, $\overline{\sigma}$ is a behavior of the bounded channel

iff for every behavior σ satisfying $Spec_B$, the behavior $\overline{\sigma}$ satisfies $Spec_C$

iff for every behavior σ satisfying $Spec_B$, the behavior σ satisfies $\overline{Spec_C}$

iff for every behavior σ, if σ satisfies $Spec_B$ then σ satisfies $\overline{Spec_C}$

iff $Spec_B \Rightarrow \overline{Spec_C}$ is true for all behaviors.

Thus, the assertion that the bounded buffer implements the bounded channel under the refinement mapping $ch \leftarrow \overline{ch}$ is expressed by:

THEOREM $Spec_B \Rightarrow \overline{Spec_C}$

When we say that the bounded buffer algorithm implements the bounded channel, we really mean that the bounded buffer algorithm's *specification* implements the bounded channel's *specification*. Implementation means implementation under a refinement mapping. When we say that one system implements another, we mean that the specification of the first system implements the specification of the second system under an appropriate refinement mapping. Generalizing the definition of implementation under a refinement mapping to an arbitrary pair of specifications is straightforward. However, this is an important concept, so you should read the following definition carefully and make sure you understand it.

Suppose we have a specification $\mathcal{H}$ with variables $x_1, \ldots, x_m$ and another specification $\mathcal{L}$ with variables $y_1, \ldots, y_n$. Think of $\mathcal{H}$ as the high-level specification and $\mathcal{L}$ as the lower-level specification or implementation. In our example, $\mathcal{H}$ is the bounded channel specification, $m = 1$, and x_1 is ch; while $\mathcal{L}$ is the bounded buffer specification, $n = 3$, and y_1, y_2, and y_3 are the three variables p, c, and buf. Let an $\mathcal{H}$-state be an assignment of values to the variables $x_1, \ldots, x_m$ and an $\mathcal{L}$-state be an assignment of values to the variables $y_1, \ldots, y_n$. A *refinement mapping* from $\mathcal{L}$-states to $\mathcal{H}$-states consists of m assignments $x_1 \leftarrow \overline{x_1}, \ldots, x_m \leftarrow \overline{x_m}$, where each $\overline{x_i}$ is an expression that may contain only constants and the variables $y_1, \ldots, y_n$. The refinement mapping maps each $\mathcal{L}$-state s to the $\mathcal{H}$-state $\overline{s}$ that assigns to each variable x_i the value of $\overline{x_i}$ in state s. This state mapping is extended to a mapping from every sequence σ of $\mathcal{L}$-states to the sequence $\overline{\sigma}$ of $\mathcal{H}$-states in the obvious way, mapping

$$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$$

to

$$\overline{s_1} \rightarrow \overline{s_2} \rightarrow \overline{s_3} \rightarrow \cdots$$

We say that $\mathcal{L}$ implements $\mathcal{H}$ under the refinement mapping $x_1 \leftarrow \overline{x_1}, \ldots, x_m \leftarrow \overline{x_m}$ iff, for every behavior σ satisfying $\mathcal{L}$, the behavior $\overline{\sigma}$ satisfies $\mathcal{H}$. This is true iff the formula $\mathcal{L} \Rightarrow \overline{\mathcal{H}}$ is true of all behaviors, where $\overline{\mathcal{H}}$ is the formula obtained from $\mathcal{H}$ by substituting the expression $\overline{x_i}$ for variable x_i, for i in $1\,..\,m$.

Remember that a property is just a temporal formula; saying that a specification *Spec* satisfies a property P just means that the formula $Spec \Rightarrow P$ is true (for all behaviors). Since $\overline{\mathcal{H}}$ is a temporal formula, it is also a property. The truth of $\mathcal{L} \Rightarrow \overline{\mathcal{H}}$ can be read either as "$\mathcal{L}$ implements $\mathcal{H}$ under the refinement mapping", or as "$\mathcal{L}$ satisfies the property $\overline{\mathcal{H}}$".

8.4.2 Showing Implementation

We now show that the specification $Spec_B$ of the bounded buffer implements the specification $Spec_C$ of the bounded channel under the refinement mapping $ch \leftarrow \overline{ch}$. This requires showing that for any behavior σ that satisfies $Spec_B$,

the behavior $\overline{\sigma}$ satisfies $Spec_C$. In the following discussion, I will forget about the $[\,]_{vars}$ and write *Next* where I should really write $[Next]_{vars}$.

A behavior σ satisfies a specification with initial predicate *Init* and next-state action *Next* iff

1. The first state of σ satisfies *Init*.
2. Every step (successive pair of states) of σ is a step that satisfies *Next*.

Let $Init_C$ and $Next_C$ be the initial predicate and next-state relations of $Spec_C$—that is, the formulas *Init* and *Next* of module *BoundedChannel* (or of module *PCalBoundedChannel*, which is equivalent). Let $Init_B$ and $Next_B$ be the corresponding formulas for $Spec_B$—that is, formulas *Init* and *Next* of module *PCalBoundedBuffer*. Conditions 1 and 2 imply that, to show that $Spec_B$ implements $Spec_C$ under the refinement mapping, we must show that:

A1. For every behavior σ whose first state satisfies $Init_B$, the first state of $\overline{\sigma}$ satisfies $Init_C$.

A2. For every behavior σ, if every step of σ satisfies $Next_B$, then every step of $\overline{\sigma}$ satisfies $Next_C$.

Recall that $\overline{\sigma}$ is obtained by changing every state s in σ to the state $\overline{s}$ that assigns the value $\overline{ch}$ to the variable ch. For any expression e that may contain the variable ch, we let $\overline{e}$ be the formula obtained by substituting $\overline{ch}$ for ch in e. This definition of overbarring an expression implies that state $\overline{s}$ satisfies $Init_C$ iff state s satisfies $\overline{Init_C}$. Condition A1 therefore asserts that for every state s, if s satisfies $Init_B$, then s satisfies $\overline{Init_C}$. In other words, we can rewrite A1 as:

B1. $Init_B \Rightarrow \overline{Init_C}$

Similarly, for every step $s \to t$ of a behavior σ, the corresponding step $\overline{s} \to \overline{t}$ of the behavior $\overline{\sigma}$ satisfies $Next_C$ iff $s \to t$ satisfies $\overline{Next_C}$. We can therefore rewrite condition A2 as:

B2. $Next_B \Rightarrow \overline{Next_C}$

Let's prove B1. From modules *BoundedChannel* and *PCalBoundedBuffer*, we have

$$Init_C \triangleq ch = \langle\,\rangle \quad \text{so} \quad \overline{Init_C} = \overline{ch} = \langle\,\rangle$$

$$\begin{aligned} Init_B \triangleq\ & \land\ buf \in [0\,..\,(N-1) \to Msg] \\ & \land\ p = 0 \\ & \land\ c = 0 \end{aligned}$$

Recall that $\overline{ch}$ is the sequence

$$\langle\ buf[c \,\%\, N],\ buf[(c \oplus 1) \,\%\, N],\ \ldots,\ buf[(p \ominus 1) \,\%\, N]\ \rangle$$

of length $p \ominus c$. If p equals c, then $p \ominus c$ equals 0 and $\overline{ch}$ is the empty sequence. Hence, $Init_B$ implies $\overline{Init}_C$, proving B1.

Let's now prove B2. From the specifications, we see that

$$\begin{array}{lcl} \overline{Next}_C & = & \overline{Send}_C \lor \overline{Rcv}_C \\ Next_B & \triangleq & Producer_B \lor Consumer_B \end{array}$$

(To make things clearer, I have continued the convention of subscripting formulas with C or B to indicate which specification they come from, even when there's no ambiguity.) We expect a $Producer_B$ step of the bounded buffer to implement a $Send_C$ step of the bounded channel, and a $Consumer_B$ step of the bounded buffer to implement a Rcv_C step of the bounded channel. Our proof of B2 should therefore be.

$\langle 1 \rangle 1.$ $Producer_B \Rightarrow \overline{Send}_C$

$\langle 1 \rangle 2.$ $Consumer_B \Rightarrow \overline{Rcv}_C$

$\langle 1 \rangle 3.$ Q.E.D.

PROOF: By $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and the definitions of $Next_B$ and $Next_C$.

Let's try proving $\langle 1 \rangle 1$. The relevant definitions are

$$\begin{array}{lcl} \overline{Send}_C & \triangleq & \land\ Len(\overline{ch}) \neq N \\ & & \land\ \exists v \in Msg : \overline{ch}' = Append(\overline{ch}, v) \end{array}$$

$$\begin{array}{lcl} Producer_B & \triangleq & \land\ p \ominus c \neq N \\ & & \land\ \exists v \in Msg : buf' = [buf \text{ EXCEPT } ![p \,\%\, N] = v] \\ & & \land\ p' = p \oplus 1 \\ & & \land\ c' = c \end{array}$$

A $\overline{Send_C}$ step appends a message to $\overline{ch}$, so it increases the length of $\overline{ch}$ by 1. Remember that the length of $\overline{ch}$ is, by definition, $p \ominus c$. Hence, a $Producer_B$ step changes the length of $\overline{ch}$ from $p \ominus c$ to $(p \oplus 1) \ominus c$. A $Producer_B$ step can be taken iff $p \ominus c \neq N$ is true. Hence, for $\langle 1 \rangle 1$ to be true, $p \ominus c \neq N$ must imply the following formula:

$$LenIncr \triangleq (p \oplus 1) \ominus c = (p \ominus c) + 1$$

Suppose c equals 3 and $p \,\%\, N$ equals $c -_N 1$, as shown in this picture. If p is a number in $0 \,..\, (2N-1)$, then this picture can show one of these two cases. Let's consider these two cases separately:

Right-Hand Case, $p = N + 2$

Recall that, for any numbers a and b, $a \oplus 1$ is the next number going clockwise around the circle, and $a \ominus b$ is the clockwise distance from b to a. Hence, $p \ominus c = N - 1$, $p \oplus 1 = N + 3$, and $(p \oplus 1) \ominus c = N$, so formula $LenIncr$ holds.

Left-Hand Case, $p = 2$

In this case, we have $p \ominus c = 2N - 1$, $p \oplus 1 = 3$, and $(p \oplus 1) \ominus c = 0$. Hence, *LenIncr* does not hold.

If we stop and think about the left-hand case, we realize that we should not have to consider it. It represents a situation in which there are $2N - 1$ messages in the buffer, and that should never happen. The problem is that we derived condition B2 from condition A2, and A2 is too strong. In fact, a little thought reveals that we could not possibly prove B2. Condition B2 asserts a condition for all possible steps (pairs of states). To prove it, we would have to show it to be true with no assumptions about the values of the variables p, c, and buf. For example, it would be have to be true even if p and c are strings rather than numbers—something we couldn't hope to prove.

Condition A2 need not be true for every behavior σ, just for behaviors that are allowed by the specification $Spec_B$. Hence, B2 need be true only for steps starting in a state that can be reached in an execution of $Spec_B$. We prove that something is true for reachable states by proving that it is true for states satisfying an invariant. We weaken B2 to the condition

$$Inv_B \Rightarrow (Next_B \Rightarrow \overline{Next_C})$$

where Inv_B is an invariant of $Spec_B$. This formula is equivalent to

B2a. $Inv_B \land Next_B \Rightarrow \overline{Next_C}$

So, to prove that $Spec_B$ implements $Spec_C$ under the refinement mapping $ch \leftarrow \overline{ch}$, we must prove B1 and B2a, for some invariant Inv.

The interesting invariant that we need is

$PCInv \triangleq p \ominus c \in 0\,..\,N$

```
PCInv == p (-) c \in 0..N
```

Add its definition to module *PCalBoundedBuffer*. As usual, the invariant must also imply type correctness. We therefore define the invariant Inv of $Spec_B$ by:

$Inv \triangleq TypeOK \land PCInv$

```
Inv == TypeOK /\ PCInv
```

Use TLC to check that Inv is an invariant of $Spec_B$.

We prove the invariance of Inv just the way we proved invariance in Section ?? for Euclid's algorithm. That is, we prove:

Inv_B is just another name for Inv that we use to remind us that it is an invariant of $Spec_B$.

1. $Init_B \Rightarrow Inv_B$
2. $Inv_B \land Next_B \Rightarrow Inv_B{}'$

Proving this, B1, and B2a proves that $Spec_B$ implements $Spec_C$ under the refinement mapping.

I will leave writing these proofs as an exercise. To illustrate how the proof is written, here is a partial proof of B2a. This proof digs down only into substep $\langle 2\rangle 2$ of step $\langle 1\rangle 1$. Moreover, it cheats in the proof of the Q.E.D. step $\langle 4\rangle 3$. Why do $\langle 3\rangle 1$, $\langle 4\rangle 1$, $\langle 4\rangle 2$, and the definition of $\overline{ch}$ imply $\langle 3\rangle 2$? At this point, if we tried to provide a more detailed proof, we'd be reduced to drawing a picture; and pictures seldom cover all cases. For example, reasoning from a picture like this is unlikely to yield a convincing argument that the result holds for $N = 1$. The problem is that we don't have a precise definition of $\overline{ch}$. The presence of the ellipsis $(\ldots)$ in our definition of $\overline{ch}$ means that it's not rigorous. Below, I will give a precise definition of $\overline{ch}$ that will permit a more rigorous proof.

Before continuing with the bounded buffer example, let's generalize what we've just done to the arbitrary refinement mapping described above. We have a specification $\mathcal{H}$ with variables $x_1, \ldots, x_m$, another specification $\mathcal{L}$ with variables $y_1, \ldots, y_n$, and a refinement mapping $x_1 \leftarrow \overline{x_1}, \ldots, x_m \leftarrow \overline{x_m}$. Let $Init_{\mathcal{H}}$ and $Next_{\mathcal{H}}$ be the initial predicate and next-state relation of $\mathcal{H}$, and let $Init_{\mathcal{L}}$ and $Next_{\mathcal{L}}$ be the corresponding formulas for $\mathcal{L}$. For any expression e, let $\overline{e}$ be the expression obtained from e by substituting $\overline{x_i}$ for x_i, for each i.

The generalizations of B1 and B2a are as follows, with the missing $[\ ]_{vars}$ constructs added:

R1. $Init_{\mathcal{L}} \Rightarrow Init_{\mathcal{H}}$

R2. $Inv_{\mathcal{L}} \wedge [Next_{\mathcal{L}}]_{\langle y_1, \ldots, y_n\rangle} \Rightarrow [\overline{Next_{\mathcal{H}}}]_{\langle \overline{x_1}, \ldots, \overline{x_m}\rangle}$

where $Inv_{\mathcal{L}}$ is an invariant of specification $\mathcal{L}$.

Let's now express all this in temporal logic. As explained above, implementation under the refinement mapping means that $\mathcal{L} \Rightarrow \overline{\mathcal{H}}$ is a theorem—that is, it is true of all behaviors. The specifications $\mathcal{H}$ and $\mathcal{L}$ are the temporal formulas defined by

$$\begin{aligned} \mathcal{H} &\triangleq Init_{\mathcal{H}} \wedge \Box[Next_{\mathcal{H}}]_{\langle x_1, \ldots, x_m\rangle} \\ \mathcal{L} &\triangleq Init_{\mathcal{L}} \wedge \Box[Next_{\mathcal{L}}]_{\langle y_1, \ldots, y_n\rangle} \end{aligned}$$

so we have

$$\overline{\mathcal{H}} = \overline{Init_{\mathcal{H}}} \wedge \Box[\overline{Next_{\mathcal{H}}}]_{\langle \overline{x_1}, \ldots, \overline{x_m}\rangle}$$

The reasoning that we use is expressed by the following proof rule. The first assumption asserts the invariance of $Inv_{\mathcal{L}}$, which is proved using rule INV. The second and third assumptions are conditions R1 and R2. The conclusion is, of

course, $\mathcal{L} \Rightarrow \overline{\mathcal{H}}$.

$$\text{REF.}\quad \frac{\begin{array}{l} Init_{\mathcal{L}} \wedge \Box[Next_{\mathcal{L}}]_{\langle y_1, \ldots, y_n\rangle} \Rightarrow \Box Inv_{\mathcal{L}}, \\ Init_{\mathcal{L}} \Rightarrow Init_{\mathcal{H}}, \\ Inv_{\mathcal{L}} \wedge [Next_{\mathcal{L}}]_{\langle y_1, \ldots, y_n\rangle} \Rightarrow [\overline{Next_{\mathcal{H}}}]_{\langle \overline{x_1}, \ldots, \overline{x_m}\rangle} \end{array}}{Init_{\mathcal{L}} \wedge \Box[Next_{\mathcal{L}}]_{\langle y_1, \ldots, y_n\rangle} \Rightarrow \overline{Init_{\mathcal{H}} \wedge \Box[Next_{\mathcal{H}}]_{\langle x_1, \ldots, x_m\rangle}}}$$

8.4.3 Expressing Implementation in TLA$^+$

We now express in TLA$^+$ the fact that the bounded buffer implements the bounded channel under our refinement mapping $ch \leftarrow \overline{ch}$. The first thing we must do is define $\overline{ch}$ in TLA$^+$. Before we can do this, you should read the section on writing functions in TLA$^{+\Box}$.

TLA$^+$ does not provide overbars, so we write *chBar* instead of $\overline{ch}$. Recall that we informally defined *chBar* to be the sequence

$$\langle\, buf[c \,\%\, N],\ buf[(c \oplus 1) \,\%\, N],\ \ldots,\ buf[(p \ominus 1) \,\%\, N] \,\rangle$$

of length $p \ominus c$. A sequence of length $p \ominus c$ is a function of the form

$$[i \in 1 \,..\, p \ominus c \mapsto \ldots]$$

so *chBar* should equal

$$[i \in 1 \,..\, p \ominus c \mapsto buf[\ldots \,\%\, N]]$$

Since $chBar[1]$ equals $buf[c \,\%\, N]$, it's not hard to see that $chBar[i]$ should equal $buf[(c \oplus (i-1)) \,\%\, N]$. The definition is therefore

$$chBar \triangleq [i \in 1 \,..\, (p \ominus c) \mapsto buf[(c \oplus (i-1))\%N]]$$ ASCII version

Add it to module *PCalBoundedBuffer*.

To prove implementation under the refinement mapping $ch \leftarrow chBar$, we reasoned about the following formulas:

- $Init_B$ and $Next_B$, which are the formulas *Init* and *Next* of module *PCalBoundedBuffer*.
- $\overline{Init_C}$ and $\overline{Next_C}$, which are the formulas obtained from formulas *Init* and *Next* of module *PCalBoundedChannel* by substituting *chBar* for *ch*.

We will express implementation in module *PCalBoundedBuffer*, so we need to express $\overline{Init_C}$ and $\overline{Next_C}$ in that module. To understand how we do this, first open module *PCalBoundedChannel* (using the Open Module command on the File menu). Observe that the module contains three declared *parameters*: the variable *ch* and the constants *N* and *Msg*. By expanding definitions, the right-hand side of every definition in that module can be written as one containing as free symbols only these parameters and the built-in symbols and constructs What's a free symbol?

of TLA$^+$. (The imported modules *Integers* and *Sequences* contain only definitions, no parameters.) For example, the definition of *Next* can be written with no defined symbols by expanding the definitions of *Send*, *Rcv*, *Len*, *Append*, and *Tail*, the last three symbols being defined in the *Sequences* module. (The symbols $=$ and $\neq$ and the constructs $\langle\,\rangle$ and $\exists v \in \ldots : \ldots$ are built into TLA$^+$.)

To express the formulas $\overline{Init_C}$ and $\overline{Next_C}$ in module *PCalBoundedBuffer*, we add the following statement to that module:

$$C \triangleq \text{INSTANCE } PCalBoundedChannel \text{ WITH } ch \leftarrow chBar,\ N \leftarrow N,\ Msg \leftarrow Msg$$

For every defined symbol F of module *PCalBoundedChannel*, this defines the symbol $C!F$ to be equal to the expression obtained as follows. What does "!" mean?

- We expand all definitions in the right-hand side of the definition of F to obtain an expression containing only the parameters *ch*, N, and *Msg* of *PCalBoundedChannel*.
- In that expression, we substitute:
  - *chBar* for *ch*.
  - The constant N of module *PCalBoundedBuffer* for N.
  - The constant *Msg* of module *PCalBoundedBuffer* for *Msg*.

Thus, it adds to module *PCalBoundedBuffer* the definition

$$C!vars \triangleq \langle chBar \rangle$$

and a definition equivalent to

$$\begin{aligned} C!Send \triangleq\ & \land\ Len(chBar) \neq N \\ & \land\ \exists\, v \in Msg : chBar' = Append(chBar,\ v) \end{aligned}$$

except that the definitions of *Len* and *Append* have been expanded. (Thus, it correctly defines $C!Send$ even though *PCalBoundedBuffer* does not import the *Sequences* module, so *Len* and *Append* are not defined.)

We can now remove the overbars from our partial proof of B2a. Here is a revised proof that does this and also partially expands the proof of step $\langle 4 \rangle 3$.

TLA$^+$ has the convention that if an INSTANCE statement contains no substitution for a parameter *Id*, then the substitution $Id \leftarrow Id$ is assumed. (The identifier *Id* must be defined or declared in the current module.) Hence, the INSTANCE statement can be written as

$$C \triangleq \text{INSTANCE } PCalBoundedChannel \text{ WITH } ch \leftarrow chBar$$

Add this statement to module *PCalBoundedBuffer*.

We could have used any identifier in place of C. Also, a parameter can be instantiated by any expression, not just an identifier. We could therefore have replaced *chBar* by its definition in the INSTANCE statement.

Earlier, we expressed implementation under the refinement mapping by:

THEOREM $Spec_B \Rightarrow \overline{Spec_C}$

In the context of module *PCalBoundedBuffer*, formula $Spec_B$ is *Spec* and $\overline{Spec_C}$ is $C!Spec$. We can therefore write this theorem in module *PCalBoundedBuffer* as

THEOREM $Spec \Rightarrow C!Spec$

We can use TLC to check the correctness of this theorem. TLC does not check theorems, but it does check properties of the specification. A property of a specification is just a temporal formula implied by the specification. We can therefore check this theorem by having TLC check the property $C!Spec$. Here's how to do it.

We can also now remove the overbars from our temporal-logic formulation of refinement. We substitute the definition of the specification *Spec* for $\mathcal{L}$ and of $C!Spec$ for $\overline{\mathcal{H}}$ in rule REF to obtain the proof rule

$$\frac{\begin{array}{l} Init \wedge \Box[Next]_{vars} \Rightarrow \Box Inv\,, \\ Init \Rightarrow C!Init\,, \\ Inv \wedge [Next]_{vars} \Rightarrow [C!Next]_{C!vars} \end{array}}{Init \wedge \Box[Next]_{vars} \Rightarrow C!Init \wedge \Box[C!Next]_{C!vars}}$$

From this example, it should be clear how to express implementation under a refinement mapping for an arbitrary specification.

8.5 Adding Fairness

We now return to the specification of the bounded channel in module *PCalBoundedChannel*. It has the form $Init \wedge \Box[Next]_{vars}$. Such a specification allows behaviors that stop at any point—including the behavior that starts in a valid initial state and then takes no steps. We now add a progress requirement on the channel's behaviors.

The typical progress requirement for a channel is that any message that is sent must eventually be received. There is no requirement that any messages actually are sent. Thus, we allow a behavior in which no steps occur. However, once a step occurs (which must be a *Send* step), there must be a corresponding *Rcv* step that removes the sent message from the channel. This progress requirement is expressed by a fairness condition on the *Rcv* process.

We say that an action A is *enabled* in a state s iff there exists a state t such that $s \to t$ is an A step. A behavior σ is *weakly fair* for A iff

- σ does not end with a state in which A is enabled, and
- σ does not contain an infinite suffix τ such that A is enabled in every state of τ and τ contains no A step.

Weak fairness of A is the property that asserts that a behavior is weakly fair for A. Weak fairness of a process means weak fairness of its next-state action.

What is strong fairness?

Question 8.4 Show that weak fairness of a process is equivalent to weak fairness of every action of the process.

ANSWER

The property weak fairness of an action A is written in TLA$^+$ as $\mathrm{WF}_{vars}(A)$, where *vars* is the subscript of all the algorithm's variables. In ASCII, it is `WF_vars(A)`. The presence of the subscript indicates that the definition of weak fairness I just gave above is not quite correct. I will correct it later. For now, just ignore the subscript. To help you ignore it, I will write the formula as $\mathrm{WF}_{vars}(A)$.

For the bounded channel specification, the *Rcv* action is enabled iff the channel is nonempty. Once the channel is nonempty, it remains nonempty if there is no *Rcv* step. Hence, weak fairness of *Rcv* implies that whenever the channel is nonempty, a *Rcv* step must eventually occur. Since messages are received in the order that they are sent, this implies that every message that is sent is eventually received.

To specify weak fairness of the *Rcv* process, we put the keyword **fair** before the keyword **process**, so the process's code is

fair process (Rcv = "R") { ... }

Modify the algorithm and run the PlusCal translator. You will see that the translation now adds to formula *Spec* the conjunct $\mathrm{WF}_{vars}(Rcv)$.

Now open the specification *PCalBoundedBuffer* in the Toolbox, and once again run TLC to check the property $C!Spec$. (Make sure that your model does not override the definition of *chBar*.) TLC should now report an error. We did not add any fairness to the bounded buffer algorithm, so its specification *Spec* allows behaviors that stop at any point. TLC will display an error trace satisfying *Spec* that ends in a state with *chBar* not empty, since such a behavior does not satisfy the fairness requirement of $C!Spec$. As you did before, use the Trace Explorer to show the values of *chBar* in each state of the trace.

Now modify the bounded buffer algorithm by inserting the keyword **fair** to specify weak fairness of the *Consumer* process. The *Consumer* action implements the bounded channel's *Rcv* action under the refinement mapping, so it is enabled whenever *chBar* is nonempty. Hence the bounded buffer algorithm should now implement property $C!Spec$. Run the PlusCal translator, which conjoins the weak fairness condition $\mathrm{WF}_{vars}(Consumer)$ to formula *Spec*. Run TLC again; it should report no error.

Question 8.5 (a) How would you rewrite the PlusCal specification of the buffered channel to turn it into a bounded *stack*, in which the receiver always receives the most recently sent message?

ANSWER

(b) Explain why fairness of the *Rcv* process in that specification does not imply that every message sent is eventually received.

Recall that specifying fairness for Euclid's algorithm conjoined to the specification the formula $\text{WF}_{vars}(Next)$. This asserts weak fairness of the entire next-state action. Make sure that you understand why the definition of weak fairness implies that this specification is not satisfied by any behavior that ends in a state with $x \neq y$.

A property is a temporal formula. Since a specification is a temporal formula, it is also a property. The properties that we use to specify systems have the form

$$Init \land \Box[Next]_{vars} \land L$$

where L is in general a conjunction of zero or more fairness formulas such as $\text{WF}_{vars}(Rcv)$. The formula $Init \land \Box[Next]_{vars}$ is a safety property, since it is false of a behavior σ iff σ's first state doesn't satisfy $Init$ or one of its steps doesn't satisfy $Next$. The formula L is a liveness property because you can't tell if a fairness property is violated by an infinite behavior just by looking at a finite prefix of the behavior. We call $Init \land \Box[Next]_{vars}$ the safety part of the specification, and we call L its liveness or fairness part.

The conjunction of 0 formulas is defined to equal TRUE. The conjunction of 1 formula equals that formula.

8.6 A Finer-Grained Bounded Buffer

We have written the bounded buffer algorithm so each of its steps implements one step of the bounded channel. Thus, in one step, the producer (1) evaluates the test $p \ominus c \neq N$, (2) assigns a value to $buf[p \,\%\, N]$, and (3) increments p modulo $2N$. Typically, an implementation is finer-grained than its specification, taking more steps. So, let's write a version of the bounded channel specification in which these are three operations are performed as separate steps. I'll discuss in Section 8.8 below how one chooses the grain of atomicity for a specification.

Recall that in a PlusCal algorithm, a step consists of an execution from one label to the next. To write a finer-grained algorithm—one that takes more steps—we just have to add more labels. Create a new specification with root a module named *FGBoundedBuffer* in the same folder (directory) as *PCalBoundedBuffer* and copy the contents of that module into it. Modify the PlusCal algorithm as follows.

- Rename the algorithm to *FGBBuf*.
- Modify the *Producer* process by adding the label $p2$ to the **with** statement and the label $p3$ to its last assignment statement. A single iteration of its **while** loop is now executed as three steps.
- Modify the *Consumer* process by adding the label $c3$ to its assignment statement. A single iteration of its **while** loop is now executed as two steps.

Here is what the algorithm should look like.

We could have added labels to the **await** statements as well, adding an extra step to the execution of a single iteration of each **while** loop. However, that extra step would just perform the trivial operation of testing if TRUE is true.

Run the PlusCal translator and look at the TLA$^+$ translation. As in the PlusCal version of Euclid's algorithm, the translation has added the variable *pc* to indicate the program control state—that is, where each process is in the execution of its code. During an execution, the value of *pc* is a function whose domain is the set {"P", "C"} of process identifiers. (The translation defines *ProcSet* to equals this set.) The value of *pc*["P"] specifies the next piece of code to be executed by the producer; for example, *pc*["P"] equals "p3" iff the next step of the producer is performed by executing the assignment to *p*. Similarly, *pc*["C"] indicates the control state of the consumer.

The definition of the initial predicate *Init* should be clear, except for the CASE statement in the last conjunct. You can easily infer the meaning of this particular CASE statement because initially *pc*["P"] should equal "p1" and *pc*["C"] should equal "c1". See Section 16.1□ for an explanation of the CASE construct.

The translation defines an action formula for each of the algorithm's labels. That formula defines the step performed by the process when control is at that label. For example, action *p*1 describes the execution of the producer's

A note about names.

await $p \ominus c \neq N$

statement. A conjunct with no primes is an enabling condition for the action. Thus, the *p*1 action is enabled iff the formulas *pc*["P"] = "p1" and $p \ominus c \neq N$ are both true. You should make sure you understand each of these action definitions.

Each process's next-state action is the disjunction of all the individual actions that describe its possible steps. The next-state action *Next* is the disjunction of the two processes' actions.

You should have copied into module *FGBoundedBuffer* the definition of the invariant *Inv* from module *PCalBoundedBuffer*. Check if you've made any mistakes by creating a small model ($N = 4$ and *Msg* a set of three elements) and running TLC to see that *Inv* is an invariant of the algorithm.

Module *FGBoundedBuffer* should also have the definition of *chBar* and the INSTANCE statement that instantiates module *PCalBoundedBuffer*. This statement defines $C!Spec$ to be the formula describing the bounded channel under the refinement mapping $ch \leftarrow chBar$. The fine-grained bounded buffer algorithm satisfies this property iff it implements the bounded channel under this refinement mapping. In the fine-grained algorithm, sending a message is implemented in three steps. The bounded channel algorithm sends a message in a single step. Three steps aren't the same as one step, so the fine-grained algorithm should not satisfy property $C!Spec$. Have TLC check this property on your model to see what error it reports.

8.6.1 Stuttering Steps

Surprise! TLC reports no error. This isn't a problem with TLC or with your model. The fine-grained bounded buffer algorithm does implement the coarser-grained bounded channel algorithm under this refinement mapping. How is that possible?

Let's use the Toolbox to see what's going on. First, we display the beginning of a behavior by adding an invariant that will be violated. The length of *chBar* is $p \ominus c$, so let TLC check the invariant

$$p \ominus c < N$$

which will be violated when N messages have been produced that have not been consumed. This produces a 13-state error trace. Let's call that 13-state behavior σ. See what values *chBar* has in the states of σ by adding the expression *chBar* in the Error-Trace Exploration section and clicking on the Explore button. Let's look at the behavior $\overline{\sigma}$ whose states are obtained by assigning to the variable *ch* the value of *chBar* in the corresponding states of σ. Here is the behavior $\overline{\sigma}$ I get, where $m1$ is an element of the set of model values substituted for *Msg* in the model I used:

$$[\, ch = \langle\,\rangle\,] \to [\, ch = \langle\,\rangle\,] \to [\, ch = \langle\,\rangle\,] \to [\, ch = \langle m1\rangle\,] \to [\, ch = \langle m1\rangle\,] \to [\, ch = \langle m1\rangle\,] \to$$
$$[\, ch = \langle m1,\ m1\rangle\,] \to [\, ch = \langle m1,\ m1\rangle\,] \to [\, ch = \langle m1,\ m1\rangle\,] \to [\, ch = \langle m1,\ m1,\ m1\rangle\,] \to$$
$$[\, ch = \langle m1,\ m1,\ m1\rangle\,] \to [\, ch = \langle m1,\ m1,\ m1\rangle\,] \to [\, ch = \langle m1,\ m1,\ m1,\ m1\rangle\,]$$

Since TLC found no error (before we added the bogus invariant), this behavior is the beginning of one that satisfies the specification *Spec* of module *PCalBoundedChannel*, which is our (coarse-grained) bounded channel specification. Why does it?

The answer to this question lies in the mysterious subscript in the formula $Init \land \Box[Next]_{vars}$. From what we learned in Section **??**, the formula is true of a behavior iff

- The behavior's first state satisfies *Init*, and
- every step of the behavior satisfies $[Next]_{vars}$.

TLA$^+$ defines $[Next]_{vars}$ to equal

$$Next \ \lor\ (\text{UNCHANGED } vars)$$

We saw above that when *vars* is defined to be a tuple of variables, then UNCHANGED *vars* asserts that all those variables are unchanged. Hence, if *vars* is the tuple of all the system's variables, then a step satisfies $[Next]_{vars}$ iff it either satisfies *Next* or it leaves all the system's variables unchanged. A step that leaves all the system's variables unchanged is called a *stuttering step*. Hence, the behavior $\overline{\sigma}$ satisfies the safety part $Init \land \Box[Next]_{vars}$ of the bounded channel's specification iff

- The initial state $[\, ch = \langle\,\rangle \,]$ of $\overline{\sigma}$ satisfies *Init*.
- Each step of $\overline{\sigma}$ either satisfies *Next* or is a stuttering step.

You should be able to see that these conditions are satisfied for this particular behavior σ, so $\overline{\sigma}$ does satisfy $Init \wedge \Box[Next]_{vars}$.

8.6.2 Why Allow Stuttering Steps?

Text has been removed from this section.

Question 8.6 (a) Reverse the two statements ANSWER

> **with** $(v \in Msg)$ { $buf[p \,\%\, N] := v$ };
> $p := p \oplus 1$

in the PlusCal algorithm of module *PCalBoundedBuffer* to get

$p := p \oplus 1$	`p := p (+) 1 ;`
with $(v \in Msg)$ { $buf[(p \ominus 1) \,\%\, N] := v$ }	`with (v \in Msg) { buf[(p (-) 1) % N] := v }`

(Note the replacement of p by $p \ominus 1$ in the second statement of the new version.) Explain why the TLA$^+$ translation of this version is equivalent to the translation of the original algorithm.

(b) Check that making the corresponding change to those two statements in module *FGBoundedBuffer* produces a specification that no longer satisfies the bounded channel specification under the refinement mapping $ch \leftarrow chBar$. Use the Trace Explorer to see why, for the error trace σ found by TLC, the behavior $\overline{\sigma}$ does not satisfy the bounded channel specification.

(c) Use TLC to check that this algorithm implements the bounded channel under the refinement mapping $ch \leftarrow \langle\,\rangle$, and explain why it does.

(d) Show that the algorithm does not implement the bounded channel under any refinement mapping in which (i) an iteration of the producer's **while** loop implements the sending of the message that the producer puts in the buffer, and (ii) an iteration of the consumer's **while** loop implements the receiving of a message.

(e) Show that (d) is not true, and find such a suitable refinement, if we weaken condition (i) by not requiring that the message sent by the bounded channel's sender is the one the producer puts in the buffer.

The following problem shows that invariance can be viewed as a trivial case of implementation under a refinement mapping. It implies that anything we say about proving implementation under a refinement mapping applies to proving invariance as well.

Question 8.7 Let *AlwaysTrue* be the specification defined as follows, where *correct* is declared to be a VARIABLE:

$$AlwaysTrue \triangleq (correct = \text{TRUE}) \land \Box[correct' = \text{TRUE}]_{correct}$$

Explain why a formula *Inv* is an invariant of a specification *Spec* iff *Spec* implements *AlwaysTrue* under the refinement mapping $correct \leftarrow Inv$.

? ← → C I S

8.6.3 Liveness and Fairness Revisited

The safety part $Init \land \Box[Next]_{vars}$ of a specification allows a behavior that takes only stuttering steps. Only the liveness part can imply that non-stuttering steps must occur. For example, the formula $\text{WF}_{vars}(Next)$ implies that a behavior cannot end in a sequence of stuttering steps from a state in which *Next* is enabled, so a *Next* step is possible.

Actually, this is not quite correct. The formula $\text{WF}_{vars}(A)$ actually asserts fairness of the formula $A \land (vars' \neq vars)$, which describes a non-stuttering A step. This temporal formula is true of a behavior σ iff σ does not contain a suffix τ such that $A \land (vars' \neq vars)$ is enabled in every state of τ and τ contains no $A \land (vars' \neq vars)$ step. (Remember that there are only infinite behaviors, so σ and all its suffixes are infinite.)

When writing specifications in TLA$^+$, almost every action A we define does not allow stuttering steps. More precisely, for most actions A, every possible A step starting in a reachable state is a non-stuttering step. However, as we saw in the PlusCal version of Euclid's algorithm, the next-state action *Next* in the TLA$^+$ translation of a PlusCal algorithm can allow stuttering steps from a halted state. The formula $\text{WF}_{vars}(Next)$ allows a behavior that ends with an infinite number of stuttering steps starting from a halting state because no non-stuttering *Next* step is possible from such a state.

Question 8.8 Explain why a formula of the form $\text{WF}_{vars}(A)$ is insensitive to stuttering. Hint

8.7 What is a Process

It seems obvious that the bounded channel and the bounded buffer are two-process systems. It also seems obvious that sequential systems like the one-bit clock and Euclid's algorithm have just a single process. What is obvious is not always true.

Consider this PlusCal algorithm. It is a two-process algorithm whose basic structure is identical to that of the PlusCal version of the bounded channel. This algorithm describes a one-bit clock, the *Tick* process waiting until b equals 0 and setting b to 1, the *Tock* process does the inverse. In the Toolbox, create

a new specification containing this algorithm (the module needs nothing else), and run the PlusCal translator. The translation defines the initial predicate *Init* and next-state action *Next* by:

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

$$Next \triangleq Tick \lor Tock$$

where *Tick* and *Tock* are defined by:

$$Tick \triangleq \land b = 0 \\ \land b' = 1$$

$$Tock \triangleq \land b = 1 \\ \land b' = 0$$

These formulas *Init* and *Next* are equivalent to the formulas *Init*1 and *Next*1 that were the initial predicate and next-state formulas of our first specification of the one-bit clock. In other words, the specification defined by this two-process algorithm is the same as our original specification of the one-bit clock. This is the same one-bit clock that we also specified in Section **??** as a one-process (sequential) PlusCal algorithm. Expressing this more mathematically, the specification *Spec* defined by the translation of the two-process one-bit clock algorithm is equivalent to the specification *Spec* defined by the translation of our original PlusCal specification of the one-bit clock.

If we look at the TLA$^+$ specifications that represent what we think of as describing two processes, we see that the next-state action is the disjunction of two formulas, each describing the steps taken by one of the processes. In fact, one reasonable definition of a process is a disjunct of the next-state action. However, as we saw in our several specifications of the one-bit clock, there can be many different ways to write equivalent specifications. Their next-state actions need not have the same disjuncts—or even the same number of disjuncts.

We usually view a concurrent system as a collection of processes, and we tend to find that view so natural that we think that the process structure is inherent in the system. It isn't. The decomposition of a system into a particular collection of processes is just a way of viewing the system; there are often other ways of viewing it. For a number of years, it seemed completely obvious that in a multi-computer system, a process was something that was executed on a single computer. The invention of remote procedure calls made it clear that one can also describe a multi-computer system with processes whose execution moves from one computer to another.

Processes provide a way of viewing a system. They are not an innate part of the system.

Question 8.9 Write a two-process PlusCal version of Euclid's algorithm whose translation produces a specification equivalent to the one we wrote in Section 4.3 in module *Euclid*.

8.8 Choosing the Grain of Atomicity

What is the right grain of atomicity for the bounded buffer algorithm? Should we use the coarse-grained algorithm *BBuf* of module *PCalBoundedBuffer* or the finer-grained algorithm *FGBBuf* of module *FGBoundedBuffer*? Or should we use a still finer-grained algorithm?

We are not interested in specifications for their own sake. A specification is interesting because it is an abstract description of a concrete system. That system might be a computer program or a hardware component that we intend to build. We want our specification to be as simple as possible, and coarser specifications are simpler than finer-grained ones. However, we also want our specification to be a sufficiently accurate representation of the system, and a finer-grained specification more accurately describes the concrete system. Our specification should be fine enough to describe all the aspects of the concrete system that are important to us, but no finer.

Before we can begin writing a specification of a system, we must choose the variables that represent the system's state. When doing that, we also decide on approximately what the grain of atomicity should be. I chose integer-valued variables p and c because I decided that reading or writing the value of each of those variables would occur in a single atomic action. Had I wanted reading or writing a single bit of those values to be a separate atomic action, I might have made them bit-valued arrays—that is, I might have let their values be functions from a set of bit locations to $\{0, 1\}$.

When specifying a system, it's a good idea to start with the coarsest-grained specification that makes sense. There's no point writing a more complicated, finer-grained specification if a coarser-grained one will reveal errors. The coarser-grained specification is easier to write and faster to check with TLC. It is usually very easy to modify a PlusCal algorithm to make it finer-grained: you just add more labels and perhaps add some process-local variables. For example, suppose we want to turn the single atomic action

$$p3:\ p := p \oplus 1$$

of *FGBBuf* into three actions: the first reading the value of p, the second applying the $\oplus 1$ operation, and the third assigning the value to p. We just replace this statement by

$$\begin{array}{ll} p3: & t := p\ ; \\ p3a: & t := t \oplus 1\ ; \\ p3b: & p := t \end{array}$$

where t is a variable local to the process (not accessible by any other process). Process-local variables are declared right before the process's code—in this case, as follows:

```
process (Process = "P")
  variable t ;
  { p1: while (TRUE) ...
```

Let's call this new algorithm $\mathcal{F}$.

Refining the grain of atomicity in a specification written directly in TLA$^+$ is not as easy, but it is still easy enough. The hard part of writing a specification is understanding what it should say. Once you get used to the language, expressing it in TLA$^+$ or PlusCal is straightforward.

How do you know if you understand something?

Should we check the finer-grained bounded buffer algorithm $\mathcal{F}$ obtained by splitting the single atomic statement $p3$ into the three separate statements $p3$, $p3a$, and $p3b$? This algorithm is a more accurate representation of an eventual implementation of the algorithm by a multi-threaded program. However, having checked the coarser-grained algorithm *FGBBuf*, do we have to check $\mathcal{F}$? The answer is no, because correctness of *FGBBuf* implies the correctness of $\mathcal{F}$. The following is a very informal proof of this claim; don't take it too seriously. The rigorous way to prove the claim is described in Section 19□.

To understand the proof, recall that correctness of an algorithm means that each of its behaviors is correct—that is, satisfies the algorithm's correctness property. For *FGBBuf*, the correctness property is $C!Spec$ of module *FGBoundedBuffer*, which asserts of a behavior τ that it is mapped by the refinement mapping $ch \leftarrow chBar$ to a behavior $\overline{\tau}$ that satisfies the bounded channel specification.

⟨1⟩1. Let $\mathcal{F}^R$ be the algorithm obtained from $\mathcal{F}$ by combining the three separate statements into the single statement:

$p3$: $t := p \;;\; t := t \oplus 1 \;;\; p := t$

I write the statement $p3$ of $\mathcal{F}^R$ in red to distinguish it from the statement $p3$ of $\mathcal{F}$.

⟨1⟩2. Correctness of algorithm *FGBBuf* implies correctness of algorithm $\mathcal{F}^R$.

PROOF: Algorithm $\mathcal{F}^R$ is the same as except that it also specifies the values of the variable t, so a behavior of $\mathcal{F}^R$ is a behavior of *FGBBuf*. Hence, correctness of every behavior of *FGBBuf* implies correctness of every behavior of $\mathcal{F}^R$.

⟨1⟩3. Correctness of $\mathcal{F}^R$ implies correctness of $\mathcal{F}$.

⟨2⟩1. It suffices to assume σ is a behavior of $\mathcal{F}$ and show that there exists a behavior σ^R of $\mathcal{F}^R$ such that correctness of σ^R implies correctness of σ.

PROOF: By definition of correctness of an algorithm.

⟨2⟩2. Define σ^R to be obtained by executing the same sequence of actions as in σ except with each $p3$ and $p3a$ step replaced by a stuttering step and each execution of $p3b$ replaced by an execution of $p3$.

$p3$ is the action of $\mathcal{F}^R$ defined in ⟨1⟩1.

PROOF: Such a behavior σ^R exists because the producer actions $p3$ and $p3a$ of $\mathcal{F}$ write only the value of t, which is never accessed by the consumer, and they read only the values of t and p, which are never written by the

consumer. Hence, these actions do not interact with any consumer actions—that is, they neither affect nor are affected by any consumer actions. It is therefore possible to defer executions of $p3$ and $p3a$ actions and combine them with the execution of the next $p3b$ action.

$\langle 2 \rangle 3$. σ^R is a behavior of $\mathcal{F}^R$

PROOF: Each step of σ^R either satisfies an action that is the same in both $\mathcal{F}$ and $\mathcal{F}^R$, is a $p3$ step, or is a stuttering step.

$\langle 2 \rangle 4$. $\overline{\sigma^R} = \overline{\sigma}$, where σ^R and σ are mapped to $\overline{\sigma^R}$ and $\overline{\sigma}$, respectively, by the refinement mapping $ch \leftarrow chBar$.

PROOF: The value of $chBar$ depends only on the values of the variables p, c, and buf. The actions $p3$ and $p3a$ of $\mathcal{F}$ leave those variable unchanged, and action $p3$ of $\mathcal{F}^R$ changes them the same way as the corresponding action $p3b$ of $\mathcal{F}$ does (incrementing p and leaving c and buf unchanged). Therefore, in each state of σ, the value of each of these three variables is the same as its value in the corresponding state of σ^R. Hence, σ and σ^R are mapped to the same behavior by the refinement mapping.

$\langle 2 \rangle 5$. Q.E.D.

PROOF: By $\langle 2 \rangle 1$, $\langle 2 \rangle 3$, and $\langle 2 \rangle 4$, since correctness of a behavior means that it is mapped by the refinement mapping $ch \leftarrow chBar$ to a behavior of the bounded channel.

$\langle 1 \rangle 4$. Q.E.D.

By $\langle 1 \rangle 2$ and $\langle 1 \rangle 3$.

This argument is quite informal. Making it more rigorous requires defining precisely what it means for two actions not to interact with one another. We could make a simple syntactic definition for PlusCal algorithms in terms of the variables that appear in both actions. However, such a definition would be too strong. For example, it would assert that

$$a\colon\ x[i] := 1 \quad \text{and} \quad b\colon\ x[j] := 2$$

interact because they both write the variable x. However, these actions do not really interact if i is unequal to j. A more precise definition is:

Remember that action A is enabled in state s iff there is a state t such that $s \to t$ is an A step.

Definition Two actions *do not interact* iff (i) executing them in either order produces the same result and (ii) executing either of them cannot enable or disable the other.

The procedure of deducing properties of a finer-grained algorithm from properties of a coarser-grained one is called *reduction*. We say that we obtain $\mathcal{F}^R$ by *reducing* $\mathcal{F}$.

I now generalize the process of reduction from the bounded buffer example to other PlusCal algorithms. Let $\mathcal{F}$ be a multiprocess PlusCal algorithm with a process Π containing a sequence of consecutive statements

$$r_1\colon\ R_1\ ;\ \ r_2\colon\ R_2\ ;\ \ \ldots\ \ r_n\colon\ R_n$$

where each R_i is a statement or sequence of statements with no labels. Define $\mathcal{F}^R$ to be the PlusCal algorithm obtained from $\mathcal{F}$ by replacing this sequence of statements with

$$r_1\colon\ R_1\ ;\ \ R_2\ ;\ \ \ldots\ \ R_n$$

In the example above, n equals 3, step r_1 is $p3$, step r_2 is $p3a$, and step r_3 is $p3b$.

Assume that $\mathcal{F}^R$ is a legal PlusCal algorithm. The PlusCal language contains a **goto** statement. For $\mathcal{F}^R$ to be a PlusCal algorithm, $\mathcal{F}$ must contain no **goto** r_i statement for $i \neq 1$. Also, PlusCal requires a **while** statement to be labeled, so no R_i can contain a **while**—except that R_1 can be a **while** statement.

To construct a mapping from behaviors of $\mathcal{F}$ to behaviors of $\mathcal{F}^R$, we assume:

Red1 There is a k in $1\,..\,n$ such that for each $i \neq k$, action r_i does not interact with any action of any process other than Π.

For each behavior σ of $\mathcal{F}$, we define a behavior σ^R by (a) for each $i \neq k$, replacing every execution of r_i in σ with a stuttering step, and (b) replacing every execution of r_k by an execution of action r_1 of algorithm $\mathcal{F}^R$. In our example, k equals 3, so r_k is the action $p3b$. The non-interactivity assumption of $p3$ and $p3a$ were necessary but not sufficient to imply the existence of the behavior σ^R for any behavior σ.

However, examining the proof reveals that this was true only because k equaled n in this example. If σ contains an execution in which an r_k is not followed by an execution of the following action $r_{k+1}, \ldots, r_n$, then we need an additional assumption to ensure that it is possible to replace the execution of r_k by an execution of r_1. A sufficient assumption is:

Red2 For every i in $(k+1)\,..\,n$, action r_i is enabled whenever control in process Π is at r_i.

To check condition Red2, note that a PlusCal action can be disabled when control is at the action only if the action contains one of the following two kinds of statements:

- **await** P with P false
- **with** $(u \in S)$ with S the empty set

Conditions Red1 and Red2 ensure that if σ is any behavior satisfying the safety property of $\mathcal{F}$ (its initial predicate and next-state action) then σ^R satisfies the safety property of $\mathcal{F}^R$. We must show that if σ also satisfies the fairness property of $\mathcal{F}$, then σ^R satisfies the fairness property of $\mathcal{F}^R$. But first, we must state what the fairness property of $\mathcal{F}^R$ is.

In Section ??, we introduced the **fair process** construct, which asserts weak fairness of the process's next-state action. As we saw in Question 8.4, weak fairness of a process is equivalent to weak fairness of each of its actions. In PlusCal, it is possible to specify different fairness conditions—weak fairness, strong fairness, or no fairness—for different actions in a process. To ensure that σ^R satisfies the fairness property of $\mathcal{F}^R$ when σ satisfies the fairness property of $\mathcal{F}$, we require that all the actions r_i of $\mathcal{F}$ have the same fairness property, and that action r_1 of $\mathcal{F}^R$ has this fairness property. Of course, all the other actions of $\mathcal{F}^R$ must have the same fairness property as the corresponding actions of $\mathcal{F}$.

Giving the actions of $\mathcal{F}^R$ the same fairness properties as those of $\mathcal{F}$ is not enough to ensure that σ^R satisfies the fairness properties of $\mathcal{F}^R$ if σ satisfies the fairness properties of $\mathcal{F}$. Consider a single-process PlusCal algorithm $\mathcal{F}$ satisfying weak fairness and containing the following statements, where t is a variable:

r_1: **with** $(u \in 1, 2)$ { $t := u$ };
r_2: **await** $t = 2$

In the reduced algorithm $\mathcal{F}^R$, these two actions are replaced by the single action:

r_1: **with** $(u \in 1, 2)$ { $t := u$ };
await $t = 2$

The TLA$^+$ translation of this action is

$$r_1 \triangleq \land\ pc = \text{“}\mathsf{r}_1\text{”} \\ \land\ \exists u \in \{1, 2\} : t' = u; \\ \land\ t' = 2 \\ \land\ pc' = \ldots$$

The formula

$$\land\ \exists u \in \{1, 2\} : t' = u; \\ \land\ t' = 2$$

is equivalent to $t' = 2$, which shows that the statement r_1 is equivalent to the simple assignment r_1: $t := 2$.

Algorithm $\mathcal{F}$ allows a behavior σ in which statement r_1 sets t to 1 and then halts. This behavior satisfies weak fairness of the actions, since it halts with pc equal to “r_2” and the r_2 action not enabled. However, the corresponding behavior σ^R halts with pc equal to “r_1” and the r_1 action enabled, which does not satisfy weak fairness for $\mathcal{F}^R$. To ensure that σ^R satisfies the fairness condition of WF^R, we need an additional condition. The following one suffices.

Red3 For any state in which the atomic statement

$$r_1\text{: } R_1 \ ; \ R_2 \ ; \ \ldots \ R_k$$

is enabled, every execution of the sequence

$$r_1\colon R_1 \ ; \ r_2\colon R_2 \ ; \ \ldots \ r_{k-1}\colon R_{k-1}$$

of statements starting in that state leaves $r_k\colon R_k$ enabled.

Condition Red3 is vacuously true if $k = 1$.

We also want to ensure that correctness of $\mathcal{F}^R$ implies correctness of $\mathcal{F}$. The generalization of the result for the bounded buffer algorithm is straightforward. Assume a refinement mapping $x_1 \leftarrow \overline{x_1}, \ldots, x_m \leftarrow \overline{x_m}$ under which the behaviors σ and σ^R are mapped to $\overline{\sigma}$ and $\overline{\sigma^R}$, respectively. The following additional condition implies that $\overline{\sigma} = \overline{\sigma^R}$.

Red4 For each $i \neq k$ and each j, action r_i leaves $\overline{x_j}$ unchanged.

This condition implies that, if $\mathcal{F}^R$ implements some specification under this refinement mapping, then so does $\mathcal{F}$. We saw in Question 8.7 that invariance is a special case of implementation under a refinement mapping. Thus, if each r_i with $i \neq k$ leaves a predicate *Inv* unchanged, then *Inv* an invariant of $\mathcal{F}^R$ implies that it is also an invariant of $\mathcal{F}$.

I will restate these results as the following informal theorem. The theorem is informal mostly because our construction of σ^R from σ is informal. It involves replacing in σ executions of one action by another, whereas a behavior consists of a sequence of states, not executions.

> **Simple PlusCal Reduction Theorem** Let $\mathcal{F}$ be a PlusCal algorithm with a process Π containing the consecutive statements
>
> $$r_1\colon R_1 \ ; \ r_2\colon R_2 \ ; \ \ldots \ r_n\colon R_n$$
>
> where each R_i has no labels, and assume that replacing these statements with
>
> $$r_1\colon R_1 \ ; \ R_2 \ ; \ \ldots \ R_n$$
>
> produces a legal PlusCal algorithm $\mathcal{F}^R$. Assume all the actions r_i of $\mathcal{F}$ have the same fairness condition, and that each action of $\mathcal{F}^R$ has the same fairness condition as the corresponding action of $\mathcal{F}$, where action r_1 of $\mathcal{F}^R$ has the same fairness condition as the actions r_i of $\mathcal{F}$. If Red1–Red3 hold, then for every behavior σ of $\mathcal{F}$ there is a behavior σ^R of $\mathcal{F}^R$ that is obtained from σ by replacing each execution of r_i by a stuttering step, for $i \neq k$, and replacing each execution of r_k by an execution of action r_1 of $\mathcal{F}^R$. Moreover, for any refinement mapping $x_1 \leftarrow \overline{x_1}, \ldots, x_m \leftarrow \overline{x_m}$, if Red4 holds, then σ and σ^R are mapped to the same behavior under this refinement mapping.

We use the reduction theorem to show that correctness of a coarser-grained algorithm implies the correctness of a finer-grained one. We start with an algorithm

$\mathcal{G}$ and split one of its actions to obtain an algorithm $\mathcal{F}$ such that $\mathcal{G}$ equals $\mathcal{F}^R$. We can continue this process by splitting another action in $\mathcal{G}$ that was copied intact into $\mathcal{F}$, obtaining a finer-grained algorithm $\mathcal{E}$ such that $\mathcal{F}$ equals $\mathcal{E}^R$; and so on.

We can apply multiple reductions to a finer-grained bounded buffer algorithm in which each of the five actions of algorithm *FGBBuf* are split. It is not obvious how the atomic actions $p1$ and $c1$ can be split into a sequence of statements. To split them, we first rewrite *FGBBuf* by using **goto** statements to eliminate the **while** loops. For example, the body of the consumer process can be written as follows:

```
c1: await p ≠ c;
c3: c := c ⊕ 1;
    goto c1
```

You should check that the rewritten algorithm has the same TLA$^+$ translation as the original, so it is obviously the same algorithm. To split the **await** statement, observe that it can be rewritten as:

```
c1: if (p = c) goto c1
```

Naively, this statement seems to be different from the original **await** statement because, when p equals c, it performs "busy waiting" steps instead of waiting. However, those steps change none of the algorithms variables, so they are just stuttering steps that are also allowed by the algorithm with the **await** statement. (The consumer can loop forever at $c1$ iff it can wait forever at the **await** statement in a behavior that is the same as the looping behavior except for stuttering steps.) Thus, we can split the consumer process's body into finer-grained actions as follows, where u is a variable local to the consumer process:

```
c1:   u := p ;
c1a: if (u = c) goto c1 ;
c3:   u := c ;
c3a: u := u ⊕ 1;
c3b: c := u;
     goto c1
```

In the reduction that combines $c1$ and $c1a$ into a single action, action $c1$ is the action r_k of the reduction theorem (in other words, k equals 1).

The splitting of the consumer actions $p1$ and $p3$ are similar. It is not clear how to split the action $p2$ that assigns a value to $buf[p \,\%\, N]$. In an application, a message might be a complex data structure, and assigning a value to $buf[p \,\%\, N]$ might consist of multiple separate writes, the last one setting it to the desired value. Just to have a definite example, let's split the $p2$ action into the following actions, where t is the variable local to the producer process that we introduced to split $p3$.

$p2$: **with** $(v \in Msg)$ { $t := v$ } ;
$p2a$: $buf[p \,\%\, N] := 42$;
$p2b$: $buf[p \,\%\, N] := t$

Each of these three actions does not interact with any consumer action. In the reduction that combines these three actions into one, we can let r_k be any of the actions.

Let $\mathcal{F}$ now be the algorithm obtained by splitting the five actions of *FGBBuf* in this way. Five reductions lead from algorithm $\mathcal{F}$ to an algorithm $\mathcal{G}$ that is the same as *FGBBuf* except for the introduction of the process-local variables t and u. These reductions can be performed in any order, and they all satisfy Red1–Red4. The only one of these fifteen conditions that is not obvious is Red4 for the reduction that "reassembles" the $p2$ action. It is not immediately clear that the assignments to $buf[p \,\%\, N]$ leave *chBar* unchanged. To see that they do, recall that *chBar* is defined to equal the sequence

$$\langle\, buf[c \,\%\, N],\ buf[(c \oplus 1) \,\%\, N],\ \ldots,\ buf[(p \ominus 1) \,\%\, N] \,\rangle$$

The invariance of *PCInv*, which asserts $p \ominus c \in 0 \,..\, N$, implies that $p \,\%\, N$ is not one of the function arguments $c \,\%\, N, \ldots, (p \ominus 1) \,\%\, N$, so changing the value of $buf[p \,\%\, N]$ does not change the value of *chBar*.

Since *FGBBuf* implements the bounded channel under the refinement mapping $ch \leftarrow chBar$, it is obvious that $\mathcal{G}$ does too. Five applications of the reduction theorem then show that $\mathcal{F}$ does as well. The reduction theorem therefore allows us to deduce the correctness of the very fine grained algorithm $\mathcal{G}$ from the correctness of *FGBBuf*.

In many real examples—perhaps most examples—Red4 does not hold, and an algorithm $\mathcal{F}$ does not implement the desired specification under the same refinement mapping as its reduced version $\mathcal{F}^R$. However, Red1–Red3 imply that $\mathcal{F}$ implements the same specification as $\mathcal{F}^R$ under a different refinement mapping that is related to the original one. This is often good enough to imply correctness of $\mathcal{F}$. Section 19□ will explain this refinement mapping and show how to formalize everything done here.