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 will 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 section 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 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 in 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 ($v \in$ BOOLEAN) { $x[self] := v$ }

Recall that the PlusCal statement **with** ($id \in S$) { Σ } 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 { Σ_1 } **or** { Σ_2 } ... **or** { Σ_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

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

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

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

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

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 that a state satisfies the invariant before checking if it satisfies the state constraint, so the constraint $p \leq 99$ wouldn't prevent the error.)

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 $q \in Nat$ 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

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

In general, the temporal operator $\leadsto$ (read *leads to* and written in ASCII as `~>`) is defined so that for any state predicates P and Q, the formula $P \leadsto Q$ 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

$$\forall \, self \in \{0,1\} : \text{WF}_{vars}(P(self))$$

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 both versions of the algorithm is the formula *Spec*, defined to equal

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

where

Init is the initial predicate.

Next is the algorithm's next-state action, which equals
$\exists \, self \in \ \in \{0, 1\} : P(self)$.

$P(self)$ is the next-state action of process *self*, which equals
$ncs(self) \vee e1(self) \vee e2(self) \vee cs(self) \vee e3(self) \vee e4(self) \vee f(self)$.

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

The specifications of the two versions are the same except for the definition of $e4$. In the version with the **while** loop, $e4$ is defined by:

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

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

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

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 $pc[i] = \text{"ncs"}$. 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 $pc[i] = \text{"ncs"}$, 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 Problem 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

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

(It's an invariant of the algorithm 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 $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$ 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 $s_t \rightarrow s_{t+1} \rightarrow \cdots$ 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 $t \geq t_1$.

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 □ formula□. This follows from the fact that the definition of WF implies that any WF formula is a □ 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: $(T0 \lor T1) \leadsto \text{FALSE}$

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: $(T0 \lor T1) \leadsto \text{FALSE}$

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:

$e1$: $x[self] := \text{TRUE}$;
  $unchecked := Procs \setminus \{self\}$;
$e2$: **while**$(unchecked \neq \{\})$
    { Set *other* to any element of *unchecked* ;
      $unchecked := unchecked \setminus \{other\}$;
      await $\neg x[other]$
    }
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

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

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:

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

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

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*.

? ← → C I S

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

$$\begin{aligned} a \prec b \;\triangleq\; & \lor\, a[1] < b[1] \\ & \lor\, (a[1] = b[1]) \land (a[2] < b[2]) \end{aligned}$$

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:

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

(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*.

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

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:

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

ASCII version

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.

$$
\begin{array}{l}
InNCS(p) \triangleq pc[p] = \text{“ncs”} \\
Waiting(p) \triangleq pc[p] = \text{“wait”} \\
FCFS \triangleq \\
\quad \forall\, p,\, q \in Procs : \\
\qquad \Box(Waiting(p) \;\wedge\; InNCS(q) \;\wedge\; \Box\neg InCS(p) \;\Rightarrow\; \Box\neg InCS(q))
\end{array}
$$

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 p has left its non-critical section but not yet set $num[p]$. To do this, we have process p indicate that it is in this part of its code by setting $flag[p]$ 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 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:

```
exit: either {  with ( k ∈ Nat ) { num[self] := k } ;
                goto exit }
   or        { num[self] := 0 }
```

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 $e1(i) \land (pc'[i] = \text{“}e2\text{”})$.

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

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.

7.9 Mutual Exclusion in Modern Programs

Most early solutions of the mutual exclusion problem assumed that reading or writing a single value is an atomic action. These algorithms could easily be implemented in real programs because, in the few multiprocess computers that existed, reading or writing a single word of memory could be modeled was an atomic action. Those days are long gone. In modern computers, each individual processor (now usually called a *core*) has its own memory cache. Reading or writing a single memory word is a complex operation that may involve communication among multiple caches and main memory. A read or write is no longer accurately modeled as an atomic action. A naive implementation of an algorithm like the one-bit algorithm does not ensure mutual exclusion. Even the method of representing reads and writes used in the bakery algorithm is not an accurate model for a modern computer. That model assumes a read or write operation is completed before the next step is executed, but a modern computer will continue executing the next program steps while actions that are part of the operation are still being performed in various parts of the memory system.

In today's multiprocess programs, processes don't communicate solely by ordinary reading and writing of memory. Modern computers provide special instructions for interprocess synchronization. These instructions make it trivial to implement mutual exclusion. Modern programming languages also provide constructs for implementing mutual exclusion. A common construct is a *lock* that can be acquired and released by a process. (Processes are usually called *threads*.) At most one process can "own" a lock; another process that tries to acquire the lock will wait until its owner releases it. Processes can use locks to execute multiple separate mutual exclusion algorithms, each with its own critical section.

The function of mutual exclusion is to make execution of a process's entire critical section act like an atomic action. Consider a multiprocess program in which each process has a single critical section. If the shared variables that are accessed in each process's critical section are accessed by other processes only in their critical sections, then an execution of each critical section can be considered to be an atomic action. Here is a more precise statement of what "can be considered to be an atomic action" means: Let Ψ be the the original program and let Π be the program with the critical sections made atomic actions. Program Ψ implements program Π under a refinement mapping such that, for every variable v of the two programs, $\overline{v}$ equals v in every state in which no process is in its critical section. (For the variable pc of Π such that $pc[p]$ represents the location of process p in its code, when p is in its critical section, $\overline{pc}[p]$ equals the control point either at or immediately after the atomic action that executes the critical section.)

The generalization to processes that execute multiple mutual exclusion algorithms is straightforward. Here is the condition that allows every critical section to be considered an atomic action: For each critical section of each process p,

the shared variables accessed in that critical section cannot be accessed by any other process while p is executing the critical section.

If all shared variables are accessed only in critical sections and we can consider those critical sections to be atomic actions, then we can apply the Single Access Rule.

I have been discussing programs as if they were precisely described by TLA$^+$ specifications. In principle, any discrete system, including a multiprocess program, can be accurately described by such a specification. In practice, that's seldom the case. The meaning of the C statement `x = x+1` in a real multiprocess program executed on a real operating system running on a real computer would be quite complicated (assuming it even had a precise meaning). In an accurate model of the program, an execution of the statement would probably consist of dozens of steps. An accurate model of an arbitrary non-trivial multiprocess program would be impossible.

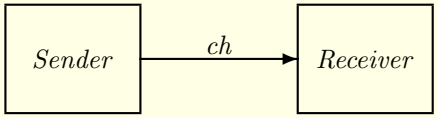
For this reason, we don't write arbitrary multiprocess programs. We write programs that access shared variables only in critical sections, so we can consider those critical sections to be atomic actions, and we can then apply the Single Access Rule. In this way, we can obtain an accurate model of the program—a model that we can describe precisely in TLA$^+$ (or PlusCal). A sensible programmer thinks about the program in terms of such a model even if she doesn't write it explicitly.

8 The Bounded Channel and Bounded Buffer

8.1 The Bounded Channel

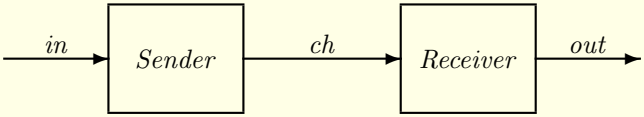
8.1.1 The Specification

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 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. We use a variable *ch* to represent the sequence of messages in the channel. We might draw a picture of the system like this:



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

To model the sender and receiver, we pretend that the sequence of all messages that the sender will send is known in advance. Let's call it *Input*. We add a variable *in* that holds the sequence of messages that the sender has yet to send, initially equal to *Input*; and a variable *out* that holds the sequence of messages that the receiver has received thus far, initially equal to the empty sequence. Our picture then becomes:



To model a system that runs forever, we let *Input* be an infinite sequence of messages.

Here is the specification of the bounded channel in pseudo-PlusCal. The processes are named *Send* and *Rcv* and have identifiers 0 and 1, respectively. To turn this pseudo-code into real PlusCal code, we need to know how to represent operations on sequences.

In TLA$^+$, a finite sequence is the same as a tuple. Finite sequences/tuples are explained in Section 15.1 and Section 15.5. Therefore, the empty sequence is the empty tuple $\langle\,\rangle$, so the variable declarations can be written:

variables $in = Input,\ out = \langle\,\rangle,\ ch = \langle\,\rangle$ **;**

For writing the spec, you will need to know only these operators that are defined in the standard *Sequences* module: *Seq*, *Len*, *Append*, *Head*, and *Tail*. Since *ch* and *out* are finite sequences, we can write the body of the receiver's **while** loop as:

```
await Len(ch) ≠ 0 ;
out := Append(out, Head(ch)) ;
ch := Tail(ch)
```

If *in* were a finite sequence, we could write the body of the sender's **while** loop as:

```
await Len(ch) ≠ N ;
ch := Append(ch, Head(in)) ;
in := Tail(in)
```

Since *in* is an infinite sequence, we need to replace *Head* and *Tail* with operators to take the head and tail of an infinite sequence. To do that, we have to decide how to represent an infinite sequence in terms of the mathematical sets and operators we have at our disposal.

A length-k sequence s is a function with domain $1 \,..\, k$, where $s[i]$ is the i^{th} element of s. Let's define infinite sequences so $s[i]$ is also the i^{th} element of an infinite sequence s. This means that an infinite sequence should be a function whose domain is the set of positive integers—a set we can write $Nat \setminus \{0\}$. Therefore, we define the set $ISeq(S)$ of all infinite sequences of elements in a set S by

$$ISeq(S) \triangleq [Nat \setminus \{0\} \to S]$$

We can then define the operators *IHead* and *ITail*, the *head* and *tail* operators for infinite sequences, by

$$IHead(iseq) \triangleq iseq[1]$$
$$ITail(iseq) \triangleq [i \in (Nat \setminus \{0\}) \mapsto iseq[i+1]]$$

Compare these definitions with the definitions of *Head* and *Tail* in the Sequences module. The definition of *IHead* is identical to that of *Head*, but it makes the specification a little easier to understand if we write $Head(s)$ when s is a finite sequence and $IHead(s)$ when it is an infinite sequence.

Let's now use TLC to check that we haven't made any errors. Using this ASCII version of the specification, open the spec in the Toolbox and run the PlusCal translator on it. To run TLC, we have to create a model. This requires specifying values for the constants N, *Msg*, and *Input*. This is easy for N and *Msg*; let N equal a small integer (say 4), and let *Msg* equal some small set of model values (say $\{m1, m2, m3\}$). But what value should we choose for *Input*, which must be an infinite sequence of messages. There are infinitely many choices—for example, $\langle m1, m2, m1, m2, \ldots \rangle$, which can be written:

```
[i ∈ Nat \ {0} ↦                          [i \in Nat \ {0} |->
   IF i % 2 = 1 THEN m1 ELSE  m2]            IF i % 2 = 1 THEN m1 ELSE m2]
```

However, you should by now have enough of a feeling for how TLC works to realize that it's unlikely to be able to handle a spec in which the value of a variable is an infinite sequence. Give it try and see what error TLC produces.

To let TLC model check the spec, we create a model that substitutes finite objects for infinite sequences. The obvious approach is to substitute finite sequences for infinite ones. We do this by overriding the definitions of *ISeq* and *ITail* by *Seq* and *Tail*, respectively. (There's no need to override the definition of *IHead* because it's equivalent to the definition of *Head*.) You can then let the model assign to *Input* any short finite sequence of messages. Do that and run TLC on the model. This should produce an error message that ends approximately as follows:

```
The error occurred when TLC was evaluating the nested
expressions at the following positions:
0.  Line 41, column 12 to line 69, column 22 in BoundedChannel
1.  Line 42, column 12 to line 70, column 38 in BoundedChannel
2.  Line 42, column 18 to line 70, column 38 in BoundedChannel
3.  Line 42, column 29 to line 70, column 37 in BoundedChannel
4.  Line 8, column 16 to line 8, column 22 in BoundedChannel
```

You can probably figure out why the error happened by reading the beginning of the message. But let's suppose you can't. Click on the last line (numbered 4) in the message. It highlights and jumps to the expression *iseq*[1] in the definition of *IHead*. This tells you that the error occurred when evaluating *IHead*(*iseq*) for some expression *iseq*. Clicking on the preceding line (numbered 3). Shows you that this evaluation occurred when TLC was evaluating the *Send* action, and *iseq* equaled *in*. (The definition of *Send* is part of the TLA$^+$ translation of the PlusCal code. If you hold the Control key down when clicking on line 3, it will take you to the occurrence of that expression in the PlusCal code.) TLC evaluates *Send* when trying to find a possible next state. The error occurred when TLC was doing this for the last state in the error-trace. The value of *in* in that state is the empty sequence $\langle\ \rangle$. The error occurred because TLC was evaluating *Head*($\langle\ \rangle$), which it obviously doesn't know how to do since TLA$^+$ doesn't specify what that value is.

Examining the error trace, we see that the error occurred because the sender had already sent all the messages in *Input*—something that could not happen if *Input* were an infinite sequence, like it's supposed to be. Since we want to find all the errors we can, we want to prevent TLC from being stopped by this error. To do that, we tell TLC not to look for any successor states to a state with *in* equal to $\langle\ \rangle$. As we've seen before, we do this by adding the State Constraint $in \neq \langle\ \rangle$ on the model's Advanced Options page. TLC should now report no errors.

8.1.2 Safety

We regard module *BoundedChannel* as the specification of the bounded channel. A specification is essentially a definition, and we cannot prove the correctness of a definition. However, we can check that a specification means what we think it does by checking that it satisfies properties we expect it to satisfy. The first property we should usually check is type correctness. The type invariant for the *BoundedChannel* spec is:

$$\begin{array}{lll} \mathit{TypeOK} & \triangleq & \land\ \mathit{in} \in \mathit{ISeq}(\mathit{Msg}) \\ & & \land\ \mathit{out} \in \mathit{Seq}(\mathit{Msg}) \\ & & \land\ \mathit{ch} \in \mathit{Seq}(\mathit{Msg}) \end{array}$$

```
TypeOK == /\ in \in ISeq(Msg)
          /\ out \in Seq(Msg)
          /\ ch \in Seq(Msg)
```

Add that definition to the module and have TLC check that *TypeOK* is an invariant of the spec.

The interesting safety property we want to check is that the receiver receives the correct messages. This means that the sequence *out* of received messages is an initial part of the infinite sequence *Input*. This requirement is expressed by the invariance of the following state predicate:

$$\mathit{CorrectReceipt} \triangleq \forall i \in 1..\mathit{Len}(\mathit{out}) : \mathit{out}[i] = \mathit{Input}[i]$$

Another invariance property we want is that there are always at most N messages that have been sent but not received. This invariant and the invariance of *CorrectReceipt* are implied by the invariance of the following assertion:

$\land\ \mathit{Len}(\mathit{inv}) \leq N$
$\land$ *Init* equals the concatenation of *out*, *ch*, and *in*

We can't write the second conjunct mathematically yet because *in* is an infinite sequence and our concatenation operator $\circ$ is defined only for finite sequences. The only kind of concatenation that makes sense for infinite sequences is the concatenation of a finite sequence and an infinite one. We define the operator $**$ as follows so $\mathit{seq} ** \mathit{iseq}$ is the concatenation of the finite sequence *seq* and the infinite sequence *iseq*. (Compare this with the definition of $\circ$ in the Sequences module.)

$$\begin{array}{l} \mathit{seq} ** \mathit{iseq} \triangleq [i \in (\mathit{Nat} \setminus \{0\}) \mapsto \\ \qquad \text{IF } i \leq \mathit{Len}(\mathit{seq}) \text{ THEN } \mathit{seq}[i] \\ \qquad\qquad\qquad \text{ELSE } \mathit{iseq}[i - \mathit{Len}(\mathit{seq})]] \end{array}$$

Using $**$ to express the invariant stated informally above, and combining it with type correctness, we obtain the following invariant:

$$\begin{array}{lll} \mathit{Inv} & \triangleq & \land\ \mathit{TypeOK} \\ & & \land\ \mathit{Len}(\mathit{ch}) \leq N \\ & & \land\ \mathit{Input} = (\mathit{out} \circ \mathit{ch}) ** \mathit{in} \end{array}$$

Add the ASCII version of the definitions of $**$ and *Inv* to module *BoundedChannel* and have TLC check that *Inv* is an invariant.

8.1.3 Liveness

The natural liveness requirement on the channel is that any message that is sent should be received. We do not require that any message is actually sent. This requirement is expressed in TLA$^+$ by weak fairness$^\square$ of the receiver's action—the *Rcv* action defined by the algorithm's TLA$^+$ translation. It is specified in PlusCal by adding the keyword **fair** before the receiver's **process** declaration. Add it and run the translator. This adds the conjunct $\mathrm{WF}_{vars}(Rcv)$ to the specification *Spec*.

Let's check that this fairness condition really does imply that every message that's sent eventually gets received. The invariant implies only messages that are sent are received, and that they are received in the correct order. This implies that to ensure that sent messages are received, we need only ensure that if messages have been sent, then some messages are received. More precisely, it suffices to show that for every message in *ch*, another message is eventually added to *out*. This is expressed in temporal logic by requiring that if there are ever i messages in *ch* and j messages in *out*, then eventually there are $i + j$ messages in *out*:

$$Liveness \triangleq \forall i \in Nat,\ j \in 1\,..\,N : (Len(out) = i) \land (Len(ch) = j) \rightsquigarrow (Len(out) = i + j)$$

ASCII version

Of course, to get TLC to check this property, you will have to use a model that overrides the definition of *Nat*.

Question 8.1 (a) How would you rewrite the PlusCal specification of the bounded 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.

8.1.4 Implementing The Bounded Channel

The variables *in* and *out* are not meant to be implemented in a real system. Real systems don't maintain an infinite sequence of inputs to be sent, and they usually don't record the entire sequence of messages that are received. What I'm interested in implementing are the steps of adding a message to the end of *ch* and removing it from the head of *ch*. I want to implement these operations in terms of lower-level operations—ones that are closer to the primitive operations performed by computers. I am therefore looking for an algorithm that uses the same variables *in* and *out*, but refines the variable *ch*. More precisely, I want an algorithm that implements algorithm *BChan* under a refinement mapping that is the identity on *in* and *out*, meaning that $\overline{in} = in$ and $\overline{out} = out$.

Another way of saying this is that I want *in* and *out* to be the "observable" variables whose behavior must be preserved. The variable *ch* is an "internal"

variable whose only function is to help describe the behavior of *in* and *out*. An implementation is free to use other internal variables, but must maintain the same behavior of *in* and *out*.

8.2 The Bounded Buffer

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

8.2.1 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

$$\begin{aligned} a +_K b &\triangleq (a + b) \,\%\, K \\ a -_K b &\triangleq (a - b) \,\%\, K \end{aligned}$$

The symbols $+_K$ and $-_K$ can't be written in TLA$^+$, but it's convenient to use them here 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 counterclockwise 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:

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

8.2.2 The Algorithm

Our *bounded buffer* algorithm implements the bounded channel by implementing the sequence *ch* of messages with a function (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 a function with domain $0\,..\,(N-1)$. (In programming terms, *buf* is an N-element array indexed by $0\,..\,(N-1)$.)

The value of *buf* will be a function from $0\,..\,(N-1)$ to the set *Msg* of messages. This means that it will be a function with domain $0\,..\,(N-1)$ such that $buf[i] \in Msg$ for all i in its domain. The set of all such functions is written in TLA$^+$ as $[0\,..\,(N-1) \to Msg]$. Thus, our bounded buffer algorithm will satisfy the type invariant

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

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 8.2.1 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 8.2.1. 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

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

The bounded buffer algorithm works because of the following property of numbers:

BB. For any $N \in Nat \setminus \{0\}$, $c \in 0\,..\,(2N-1)$, and $j \in 0\,..\,N$, the j numbers $c\%N$, $(c +_{2N} 1)\%N$, ..., $(c +_{2N} (j-1))\%N$ are all distinct.

Property BB is true with $2N$ replaced by kN for any integer $k > 1$, so we could use kN instead of $2N$ in the algorithm. For simplicity, we use $2N$.

We can now write our algorithm. Like the bounded channel algorithm, the bounded buffer algorithm will have a variable *in* containing the infinite sequence of messages yet to be sent on the channel and a variable *out* containing the (finite) sequence of messages that have been received from the channel. We will need the same definitions of *ISeq*, *ITail*, *IHead*, and $**$ as in the bounded channel specification. We could just copy and paste them into the bounded buffer spec, but it's a little nicer to use a separate module that's imported into each specification. Create a new module *ISequences* using this ASCII text. Delete the definitions of those four operators from module *BoundedChannel* and add *ISequences* to that module's EXTENDS statement.

Now create a new spec with root module *BoundedBuffer*. The module begins just like module *BoundedChannel*.

EXTENDS *Integers*, *Sequences*, *ISequences*
CONSTANT N, *Msg*, *Input*
ASSUME $\land N \in Nat \setminus \{0\}$
$\land$ *Input* $\in$ *ISeq*(*Msg*)

```
EXTENDS Integers, Sequences
CONSTANT N, Msg, Input
ASSUME /\ N \in Nat \ {0}
       /\ Input \in ISeq(Msg)
```

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

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

```
a (+) b == (a + b) % 2*N
a (-) b == (a - b) % 2*N
```

We now come to the algorithm, which we name *BBuf*. 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. Click here to see the algorithm's code.

The **variables** statement declares the same variables *in* and *out* as in the bounded channel spec, plus 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 8.2 How would you write the declarations of p and c so they both initially equal the same arbitrary element of $0\,..\,(N-1)$? ANSWER

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 $p1$ and $c1$ because

later we will add labels $p2$, $p3$, $c2$, and $c3$. You should have no trouble understanding these processes by comparing them with the processes of the bounded channel algorithm in Section 8.1. As in the bounded channel, we require that sent messages are eventually received by making the *Consumer* a **fair** process.

Note

Here is the complete ASCII version of the *BoundedBuffer* module. Use it to complete the module in the Toolbox, and run the translator.

To check that we haven't made any simple mistakes, we should check that the spec is type correct—i.e., that is satisfies a type-correctness invariant. Here is a suitable invariant:

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

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

Use TLC to check that *TypeOK* is an invariant of the spec. As with the bounded channel spec, you will have to use a model that overrides the definitions of *ISeq* and *ITail* and that uses a state constraint to prevent TLC from reporting an error after all the elements of *Input* have been sent.

8.3 The Bounded Buffer Implements the Bounded Channel

8.3.1 The Refinement Mapping

We now show that the *BoundedBuffer* specification implements the *BoundedChannel* specification under a suitable refinement mapping. Refinement mappings and implementation were explained in Section 6.6 and Section 6.8. You may want to review those sections now.

A refinement mapping from *BoundedBuffer* to *BoundedChannel* consists of definitions of state functions $\overline{in}$, $\overline{out}$, and $\overline{ch}$ in terms of the variables of *BoundedBuffer*. Recall that for any formula F of *BoundedChannel*, we define $\overline{F}$ to be the formula obtained from F by substituting the expressions $\overline{in}$, $\overline{out}$, and $\overline{ch}$ for in, out, and ch, respectively. (The variables of $\overline{F}$ are variables of *BoundedBuffer*.)

Let Spec_C be formula *Spec* of module *BoundedChannel* and let Spec_B be formula *Spec* of module *BoundedBuffer*, so Spec_C is the specification of the bounded channel and Spec_B the specification of the bounded buffer. We say that Spec_B implements Spec_C under the refinement mapping iff the following condition is satisfied: For every behavior $s_1 \to s_2 \to \cdots$ satisfying formula Spec_B, the state sequence $\overline{s_1} \to \overline{s_2} \to \cdots$ is a behavior satisfying formula Spec_C, where each $\overline{s_i}$ is the state that assigns to the variables in, out, and ch of *BoundedChannel* the values of the state functions $\overline{in}$, $\overline{out}$, and $\overline{ch}$, respectively, in the state s_i. This condition is equivalent to the assertion that $\mathit{Spec}_B \Rightarrow \overline{\mathit{Spec}_C}$ is a theorem—that is, a formula that is true for all behaviors.

A more precise definition of $\overline{s_i}$.

As explained in Section 8.1.4, we want the refinement mapping to be the identity on *in* and *out*, so $\overline{in} = in$ and $\overline{out} = out$. This means that the behavior $s_1 \to s_2 \to \cdots$ of *BoundedBuffer* and the corresponding behavior $\overline{s_1} \to \overline{s_2} \to \cdots$ have the same values of *in* and *out*, so they represent the same sending and receipt of messages. We consider *in* and *out* to be the only observable variables, and if we look only at the observable variables *in* and *out*, then $s_1 \to s_2 \to \cdots$ and $\overline{s_1} \to \overline{s_2} \to \cdots$ are the same behavior (where two behaviors that differ only in stuttering steps are considered to be the same). Let $VSpec_C$ be specification $Spec_C$ with the internal variable *ch* hidden. Then $Spec_B$ implements $Spec_C$ under the refinement mapping iff every behavior that satisfies $Spec_B$ also satisfes $VSpec_C$—in other words, iff $Spec_B \Rightarrow VSpec_C$ is a theorem. We write $VSpec_C$ informally as $\exists\, ch : Spec_C$.

The interesting part of the refinement mapping is the definition of $\overline{ch}$. Recall that the sequence of messages in the channel *ch* is represented by the sequence of elements

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

We define $\overline{ch}$ to equal this sequence of messages. For example, in this picture, $\overline{ch}$ equals

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

To define $\overline{ch}$ precisely, recall that the number of messages in the buffer is $p \ominus c$. From this and the fact that the first message in the buffer (if there is one) is $buf[c \,\%\, N]$, the definition is straightforward. Since $\overline{ch}$ isn't a TLA$^+$ identifier, we call it *chBar*:

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

```
chBar == [i \in 1..(p (-) c) |-> buf[(c (+) (i - 1)) % N]]
```

Add this definition to module *BoundedBuffer*. To check it, create an error trace by having TLC check an incorrect invariant—for example, $Len(out) \neq 4$. Run the Trace Explorer (as you did before), using it to display the value of *chBar* in each state of the trace.

The following statement imports module *BoundedChannel* into module *BoundedBuffer* so that $C!Spec$ equals $\overline{Spec_C}$:

$$C \triangleq \text{INSTANCE } BoundedChannel \text{ WITH } ch \leftarrow chBar,\ in \leftarrow in,\ out \leftarrow out,\ N \leftarrow N,\ Msg \leftarrow Msg,\ Input \leftarrow Input$$

TLA$^+$ allows us to omit any substitution of the form $id \leftarrow id$ from the WITH clause, so we can write this statement as

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

ASCII version

In fact we could have used the identifier *ch* instead of *chBar* in module *BoundedBuffer* and eliminated the entire WITH clause.

Add the INSTANCE statement to module *BoundedBuffer*. You can now have TLC check that $Spec_B \Rightarrow \overline{Spec_C}$ is a theorem by having it check the property $C!Spec$. (Add $C!Spec$ to the Properities subsection of the What to check? part of the Model Overview page.) If you followed the instructions in Section 8.1.3 and added a fairness condition to the bounded channel spec, then TLC should report the error

```
Temporal properties were violated.
```

Since we haven't yet added any fairness condition to the bounded buffer algorithm, $Spec_B$ is satisfied by behaviors $s_1 \to s_2 \to \cdots$ for which $\overline{s_1} \to \overline{s_2} \to \cdots$ doesn't satisfy the weak fairness property of $Spec_C$. If you remove that fairness condition (by deleting the **fair** keyword and running the translator), TLC should find no error.

8.3.2 Showing Implementation

Since TLC finds no counterexample, we can try to prove $Spec_B \Rightarrow \overline{Spec_C}$. Let's drop the subscripts and write our proof as it would appear inside module *BoundedBuffer*. Remember that $C!Id$ means $\overline{Id}$ for any identifier *Id* defined in *BoundedChannel*. Any other identifier has the meaning it has in *BoundedBuffer*. Thus, we have to prove $Spec \Rightarrow C!Spec$. We saw in Section 6.8□ that to prove $Spec \Rightarrow C!Spec$, we have to prove two things:

R1. $Init \Rightarrow C!Init$

R2. $Inv \land Inv' \land Next \Rightarrow [C!Next]_{C!vars}$

for some invariant *Inv* of the bounded buffer. Let's start with R1.

Looking at the translations of the two PlusCal algorithms, and remembering that $\overline{ch} = chBar$, $\overline{in} = in$ and $\overline{out} = out$, we see that:

$$\begin{aligned} Init = \land\ & in = Input \\ \land\ & out = \langle\,\rangle \\ \land\ & buf \in [0\,..\,(N-1) \to Msg] \\ \land\ & p = 0 \\ \land\ & c = 0 \end{aligned} \qquad\qquad \begin{aligned} C!Init = \land\ & in = Input \\ \land\ & out = \langle\,\rangle \\ \land\ & chBar = \langle\,\rangle \end{aligned}$$

Obviously, *Init* implies the first two conjuncts of $C!Init$. To see that it also implies $chBar = \langle\,\rangle$, recall that *chBar* is the sequence

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

of length $p \ominus c$. Since $p = c$ implies $p \ominus c = 0$, *Init* implies $chBar = \langle\,\rangle$, proving R1. (A rigorous proof uses the third conjunct of *Init*, which implies that *buf* is a function with domain $0\,..\,(N-1)$.)

Let's now prove R2. We expect that *Inv* should imply *TypeOK*. We have defined *chBar* to be a sequence of length $p \ominus c$. Since *Spec* implies that *ch* is always a sequence of length at most N, to prove *Spec* we must be able to prove that $p \ominus c$ is at most N. Hence *Inv* must imply $p \ominus c \leq N$. It turns out that this is the only additional property we need to prove R2, so we can define:

$$Inv \triangleq TypeOK \land (p \ominus c \leq N)$$

Note that by definition of $\ominus$ and *TypeOK*, this implies $p \ominus c \in 0 \,..\, N$.

From the specifications, we see that

$$\begin{aligned} C!Next &= C!Send \lor C!Rcv \\ Next &= Producer \lor Consumer \end{aligned}$$

We expect a *Producer* step of the bounded buffer to implement a *Send* step of the bounded channel, and a *Consumer* step of the bounded buffer to implement a *Rcv* step of the bounded channel. To prove R2, we therefore prove

1. $Inv \land Inv' \land Producer \Rightarrow C!Send$
2. $Inv \land Inv' \land Consumer \Rightarrow C!Rcv$

Here is a partial proof of R2 containing the proof of property 1. It uses these definitions:

$$\begin{aligned} C!Send \triangleq\ & \land\ Len(chBar) \neq N \\ & \land\ chBar' = Append(chBar, IHead(in)) \\ & \land\ in' = ITail(in) \\ & \land\ out' = out \\ Producer \triangleq\ & \land\ p \ominus c \neq N \\ & \land\ buf' = [buf \text{ EXCEPT } ![p\%N] = IHead(in)] \\ & \land\ in' = ITail(in) \\ & \land\ p' = p \oplus 1 \\ & \land\ \text{UNCHANGED } \langle out, c \rangle \end{aligned}$$

The proof of property 2 is left as an exercise, as is the proof of invariance of *Inv*.

Question 8.3 Complete the proof of R2 and show that *Inv* is an inductive invariant of *Spec*.

8.3.3 Liveness

Our liveness requirement for the bounded channel is weak fairness of the receiver's action. The corresponding requirement for the bounded buffer is weak fairness of the consumer's action, which we assert by adding the keyword **fair**

before the producer's **process** declaration. Do that and rerun the translator. The specification *Spec* now becomes

$$Init \wedge \Box[Next]_{vars} \wedge \mathrm{WF}_{vars}(Consumer)$$

We expect that the fairness requirement of the bounded buffer should implement the fairness requirement of the bounded channel, under the refinement mapping we have been using. More precisely let's once again add to the specification *Spec* of module *BoundedChannel* the fairness conjunct $\mathrm{WF}_{vars}(Rcv)$ (by putting **fair** before its **process** declaration and running the translator). We expect $Spec \Rightarrow C!Spec$ to still be a theorem. Use TLC to check that it is. We now prove that it's a theorem.

? ← → C I S

The fairness condition of the bounded channel spec gives $C!Spec$ the conjunct $\overline{\mathrm{WF}_{vars}(Rcv)}$, the formula obtained from $\mathrm{WF}_{vars}(Rcv)$ by substituting *chBar* for *ch*. We can't write this formula in TLA$^+$. It does not equal $\mathrm{WF}_{\overline{vars}}(\overline{Rcv})$, which we can write as the TLA$^+$ formula $\mathrm{WF}_{C!vars}(C!Rcv)$. We know that $\overline{\mathrm{WF}_{vars}(Rcv)}$ equals

Any symbol appearing under a bar comes from module *BoundedChannel*.

$$\Box\Diamond \neg \overline{\mathrm{ENABLED}\ \langle Rcv \rangle_{vars}} \vee \Box\Diamond \langle C!Rcv \rangle_{C!vars}$$

but we can't write $\overline{\mathrm{ENABLED}\ \langle Rcv \rangle_{vars}}$ in TLA$^+$. Even though we can't write this formula in TLA$^+$, it is mathematically well defined and we can use it in our informal proof.

We've already proved the safety part of $Spec \Rightarrow C!Spec$—namely, the formula:

$$Init \wedge \Box[Next]_{vars} \Rightarrow \overline{Init_C \wedge \Box[Next_C]_{vars_C}}$$

We therefore just have to prove $Spec \Rightarrow \overline{\mathrm{WF}_{vars_C}(Rcv_C)}$. Before we do this, you should read (or re-read) Section 17.5.5$^{\Box}$ on proving liveness properties.

In proving the safety part, we showed that a consumer step implements a receiver step—that is, we proved that *Consumer* implies $C!Rcv$. This suggests that we should be able to use the simplest proof rule from Section 17.5.5:

$$\frac{\langle A \rangle_v \wedge \overline{\mathrm{ENABLED}\ \langle B \rangle_w} \Rightarrow \langle \overline{B} \rangle_{\overline{w}} \qquad \overline{\mathrm{ENABLED}\ \langle B \rangle_w} \Rightarrow \mathrm{ENABLED}\ \langle A \rangle_v}{\mathrm{WF}_v(A) \Rightarrow \overline{\mathrm{WF}_w(B)}}$$

with the substitutions:

$$A \leftarrow Consumer \qquad v \leftarrow vars \qquad B \leftarrow Rcv_C \qquad w \leftarrow vars_C$$

Actually, we can't expect to prove either of the hypotheses without an invariant, so we extend the proof rule to make use of an invariant I of the bounded buffer

spec:

$$\text{WF2s.}\quad \frac{I \wedge I' \wedge \langle A\rangle_v \wedge \overline{\text{ENABLED}\ \langle B\rangle_w} \;\Rightarrow\; \langle \overline{B}\rangle_{\overline{w}} \qquad I \wedge \overline{\text{ENABLED}\ \langle B\rangle_w} \;\Rightarrow\; \text{ENABLED}\ \langle A\rangle_v}{\Box I \wedge \text{WF}_v(A) \;\Rightarrow\; \overline{\text{WF}_w(B)}}$$

We will determine below what invariant to substitute for I.

Question 8.4 Derive rule WF2s from rule WF2a$^{\Box}$. ANSWER

Proving the safety part required proving $Inv \wedge Inv' \wedge Consumer \Rightarrow C!Rcv$. Since a *Consumer* step modifies c and out, it's easy to prove

$$TypeOK \Rightarrow (Consumer \Rightarrow (vars' \neq vars))$$

Hence, $Inv \wedge Inv'$ (which implies $TypeOK$) implies $Consumer \equiv \langle Consumer\rangle_{vars}$. Thus, the first hypothesis of WF2s follows from what we have already proved if I implies Inv. (We don't need the $\overline{\text{ENABLED}\ \langle B\rangle_w}$ on the left of the implication to prove the first hypothesis.)

When we turn to the second hypothesis of WF2s, we face a problem: It requires reasoning about the formula $\overline{\text{ENABLED}\ \langle Rcv_C\rangle_{vars_C}}$ that we can't even write in TLA$^+$. It's actually easy to write this formula in TLA$^+$; we just add the definition

$$RcvEnabled \triangleq \text{ENABLED}\ \langle Rcv\rangle_{vars}$$

to module *BoundedChannel*, so $\overline{\text{ENABLED}\ \langle Rcv\rangle_{vars}}$ equals $C!RcvEnabled$. This doesn't help because we can't expand the definition of $C!RcvEnabled$ as a TLA$^+$ formula, but it suggests the following approach: We define *RcvEnabled* to be a formula that's equivalent to $\text{ENABLED}\ \langle Rcv\rangle_{vars}$ such that $C!RcvEnabled$ can be written in TLA$^+$. More precisely, we prove (in module *BoundedChannel*) that *RcvEnabled* is equivalent to $\text{ENABLED}\ \langle Rcv\rangle_{vars}$. The obvious definition is

$$RcvEnabled \triangleq Len(ch) \neq 0$$

In the context of module *BoundedChannel*, we must prove:

$$\text{THEOREM}\ RcvEnabledThm \triangleq TypeOK \Rightarrow (RcvEnabled \equiv \text{ENABLED}\ \langle Rcv\rangle_{vars})$$

(Here, *TypeOK* is the formula of that name defined in *BoundedChannel*.) Since substituting state functions for variables (and constant expressions for constants) in a valid formula produces a valid formula, $C!RcvEnabledThm$ is a valid theorem. It asserts:

$$C!TypeOK \Rightarrow (C!RcvEnabled \equiv \overline{\text{ENABLED}\ \langle Rcv\rangle_{vars}})$$

Thus, if I implies $C!TypeOK$, then (after expanding the definition of $C!RcvEnabled$) the second hypothesis of WF2s becomes

$$I \land (Len(chBar) \neq 0) \Rightarrow \text{ENABLED } \langle Consumer \rangle_{vars}$$

There was no need to define *RcvEnabled*. We could simply replace it with its definition, $Len(ch) \neq 0$, in *RcvEnabledThm*.

Let I equal $Inv \land C!TypeOK$. We have reduced the proof of $Spec \Rightarrow \text{WF}_{vars_C}(Rcv_C)$ to proving three things:

1. Theorem *RcvEnabledThm*.
2. $Inv \land (Len(chBar) \neq 0) \Rightarrow \text{ENABLED } \langle Consumer \rangle_{vars}$
3. $Spec \Rightarrow \Box(Inv \land C!TypeOK)$.

The proofs of 1 and 2 use rules E1–E7□ of Section 17.5.1. For 1, observe that *TypeOK* (of module *BoundedChannel*) implies that a *Rcv* action changes *ch* and *out*, so $\langle Rcv \rangle_{vars}$ is equivalent to *Rcv* and E7 implies $\text{ENABLED } \langle Rcv \rangle_{vars} \equiv \text{ENABLED } Rcv$. We then use E1, E4, and E5 to show $\text{ENABLED } Rcv \equiv Len(ch) \neq 0$, proving 1. The proof of 2 is similar (except in the context of module *BoundedBuffer*).

To prove 3, first observe that in proving safety, we proved $Spec \Rightarrow \Box Inv$. Hence, we just have to prove $Spec \Rightarrow \Box C!TypeOK$. In proving safety, we also proved

$$Spec \Rightarrow C!Init \land \Box[C!Next]_{C!vars}$$

Hence we just have to prove

$$C!Init \land \Box[C!Next]_{C!vars} \Rightarrow C!TypeOK$$

Since substitution (barring) preserves truth of a formula, it suffices to prove the following theorem in module *BoundedChannel*:

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

This is a straightforward invariance proof.

8.4 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 $2N$. As explained in Section 7.8.2, such an algorithm is not considered to be a satisfactory implementation because it doesn't satisfy the Single Access Rule.

We get a finer-grained implementation of the bounded channel by adding labels to the bounded buffer algorithm *BBuf*. That algorithm's shared variables are *buf*, *p*, and *c*. (Variable *in* is accessed only by the producer; variable *out* is accessed only by the consumer.) Moreover, *p* is written only by the producer and *c* is written only by the consumer. We get a finer-grained version that satisfies the Single Access Rule by adding labels as shown here.

Here is the ASCII text of a new module *FGBoundedBuffer* that's the same as the beginning of the *BoundedBuffer* module, except with the additional labels in the algorithm and the algorithm renamed *FGBBuf*. Open it in the Toolbox and run the translator. Create the same kind of model for it that you used for the original bounded buffer spec, and check that the algorithm satisfies the same type invariant *TypeOK*.

The TLA$^+$ translation of *FGBuf* introduces the additional variable *pc*, which satisfies this type-correctness invariant, where STRING is the set of all strings:

$pc \in [\{\text{“P”}, \text{“C”}\} \to \text{STRING}]$ `pc \in [{"P", "C"} -> STRING]`

Add it as an additional conjunct to *TypeOK*. We can make this condition more precise by adding the following two conjuncts as well:

$\land\ pc[\text{“P”}] \in \{\text{“p1”}, \text{“p2”}, \text{“p3”}\}$ `/\ pc["P"] \in {"p1", "p2", "p3"}`
$\land\ pc[\text{“C”}] \in \{\text{“c1”}, \text{“c2”}, \text{“c3”}\}$ `/\ pc["C"] \in {"c1", "c2", "c3"}`

Have TLC check that the resulting formula *TypeOK* is invariant of *FGBBuf*.

Question 8.5 We might expect algorithm *FGBBuf* to implement the bounded channel algorithm *BChan* under the same refinement mapping as algorithm *BBuf*. Use TLC to see why it doesn't. ANSWER

Instead of showing directly that *FGBBuf* implements *BChan*, we show that it implements *BBuf* under a refinement mapping

$buf \leftarrow \overline{buf} \qquad p \leftarrow \overline{p} \qquad c \leftarrow \overline{c} \qquad in \leftarrow \overline{in} \qquad out \leftarrow \overline{out}$

where we use red overbars to distinguish this refinement mapping from the refinement mapping

$ch \leftarrow \overline{ch} \qquad in \leftarrow \overline{in} \qquad out \leftarrow \overline{out}$

under which *BBuf* implements *BChan*. We know that *BBuf* implements *BChan* under the “black” refinement mapping. If *FGBuf* implements *BBuf* under the “red” refinment mapping, then we can conclude that *FGBuf* implements *BChan* under refinement mapping

$ch \leftarrow \overline{\overline{ch}} \qquad in \leftarrow \overline{\overline{in}} \qquad out \leftarrow \overline{\overline{out}}$

For any expression *e* in the variables *ch*, *in*, and *out* of *BChan*, the expression $\overline{\overline{e}}$ is obtained as follows.

1. Perform the black substitutions for the variables of *BChan* to obtain the expression $\overline{e}$ in the variables *buf*, *p*, *c*, *in*, and *out* of *BBuf*.

2. Perform the red substitutions for the variables of *BBuf* to obtain $\overline{\overline{e}}$, which is an expression in the variables *buf*, *p*, *c*, *in*, *out*, and *pc* of *FGBBuf*.

As explained in Section 8.1.4, we are letting both refinement mappings be the identity on *in* and *out*, so $\overline{\overline{in}} = in$ and $\overline{\overline{out}} = out$.

Since $\overline{in} = in$, a *Producer* step of *BBuf* must be simulated by a *p*2 step of *FGBuf*. Therefore, a *p*2 step must increment $\overline{p}$ and a *p*1 or *p*3 step must leave $\overline{p}$ unchanged. This is accomplished by letting

$$\overline{p} \triangleq \text{IF } pc[\text{“P”}] = \text{“p3”} \text{ THEN } p \oplus 1 \text{ ELSE } p$$

Similar reasoning leads to the analogous definition of $\overline{c}$. Therefore, add the following to module *FGBoundedBuffer*.

$$pBar \triangleq \text{IF } pc[\text{“P”}] = \text{“p3”} \text{ THEN } p \oplus 1 \text{ ELSE } p$$
$$cBar \triangleq \text{IF } pc[\text{“C”}] = \text{“c3”} \text{ THEN } c \oplus 1 \text{ ELSE } c$$
$$B \triangleq \text{INSTANCE } NewBoundedBuffer \text{ WITH } p \leftarrow pBar,\ c \leftarrow cBar$$

ASCII version

Let TLC check if *FGBBuf* implements *BBuf* under this refinement mapping by having it check the temporal propertyh *B*!*Spec*. TLC will report that temporal properties are violated, which means that *FGBBuf* doesn't implement the fairness properties of *BBuf* under this refinement mapping. The error trace is one in which the producer executes actions *p*1 and *p*2, and then the execution halts (stutters forever). In this last state, $p = c = 0$, so a consumer action is not enabled. The only enabled action is the producer's *p*3 action. Since there is no fairness condition on the producer, that step need not be taken, so the execution can halt.

We don't want to require that the producer keeps performing observable steps—*p*2 steps that remove an element from *in*. However, if it does perform such a step, then it must complete the operation and perform a *p*3 step. Therefore, we want to require fairness of the *p*3 action but not of the *p*2 action. We express this in PlusCal by making the producer a **fair** process, but change the *p*2 label to `p2:-`. Since a *p*1 step changes no variable except *pc*, it doesn't matter whether we require fairness of *p*1 or not require it by adding a `-` after its label. I find it more natural not to require it.

Make the necessary change to the PlusCal code, rerun the translator, and have TLC check that *FGBBuf* now implements *BBuf* under the refinement mapping. Check it without fairness of *p*1. Since adding fairness strengthens the spec, so *FGBBuf* still satisfies any properties that it did without fairness of *p*1, it will also implement *BBuf* with the fairness property.

Question 8.6 Write the refinement mapping under which *FGBBuf* implements the bounded channel specifiction *BChan*, and have TLC check that it's correct.

8.5 Further Refinement

When the producer is at $p2$, it is about to access buffer element $p \,\%\, N$. When the consumer is at $c2$, it is about to access buffer element $c \,\%\, N$. The following state predicate asserts that if both processes are about to access buffer elements, then they are about to access different elements:

$$BufMutex \triangleq (pc[\text{“P”}] = \text{“p2”}) \land (pc[\text{“C”}] = \text{“c2”}) \Rightarrow (p \,\%\, N \neq c \,\%\, N)$$

Have TLC check that *BufMutex* is an invariant of algorithm *FGBBuf*.

Formula *BufMutex* asserts a sort of mutual exclusion property. We can think of each of the two processes having N separate critical sections, numbered from 0 to $N-1$. The producer is in its i^{th} critical section when $pc[\text{“P”}]$ equals “p2” and $p \,\%\, N$ equals i; and likewise, the consumer is in its i^{th} critical section when $pc[\text{“C”}]$ equals “c2” and $c \,\%\, N$ equals i. The state predicate *BufMutex* asserts that the two processes cannot be in their i^{th} critical sections at the same time, for each i in $0\,..\,(N-1)$. If we can consider $buf[i]$ and $buf[j]$ to be separate variables if $i \neq j$, then the discussion in Section 7.9 shows that we can implement statements $p2$ and $c2$ with non-atomic actions.

Are $buf[0]$ and $buf[1]$ separate variables?

As a very simple example of refining the grain of atomicity of the $p2$ and $c2$ steps, let's just refine $p2$ into two separate atomic actions. Create a new specification by copying the beginning of module *FGBoundedBuffer* through the algorithm declaration, and changing the algorithm by replacing the two-line statement $p2$ with

$p2a$:- $buf[p\%N] := IHead(in)$;
$p2b$: $in := ITail(in)$;

```
p2a:-  buf[p % N] := IHead(in);
p2b:   in := ITail(in) ;
```

This algorithm implements *FGBBuf* under a refinement mapping for which $\overline{buf}$ and $\overline{out}$ are changed by $p2a$. They are defined as follows:

$$\overline{in} \triangleq \text{IF } pc[\text{“P”}] = \text{“p2b”} \text{ THEN } ITail(in) \text{ ELSE } in$$

$$\overline{pc} \triangleq [i \in \{\text{“P”}, \text{“C”}\} \mapsto \text{CASE } i = \text{“P”} \to \text{CASE } pc[\text{“P”}] = \text{“p2a”} \to \text{“p2”} \;\square\; pc[\text{“P”}] = \text{“p2b”} \to \text{“p3”} \;\square\; \text{OTHER} \to pc[\text{“P”}] \;\square\; i = \text{“C”} \to pc[\text{“C”}] \;]$$

ASCII version

The refinement mapping is the identity on *out*, *buf*, p, and c. Use TLC to check that the new algorithm implements *FGBBuf* under this refinement mapping.

Question 8.7 Why do we want a “-” after the label $p2a$ but not after the label $p2b$.

ANSWER

This refinement mapping is not the identity on *in*. The new algorithm also refines *FGBBuf* under a refinement mapping that is the identity on *in* (as well as *out*)—a mapping under which a $p2$ step of *FGBBuf* is implemented by a $p2b$ step. To define that refinement mapping, we must add a history variable that records the value of $buf[p \,\%\, N]$—the value that is overwritten by step $p2a$. History variables are explained in Section 18.

Question 8.8 Add the necessary history variable to the new algorithm and define the refinement mapping that is the identity on *in* and *out* under which the algorithm implements *FGBBuf*.

8.6 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 bounded channel algorithm (without fairness). Algorithm *Clock* 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:

$$Tick \triangleq \begin{array}{l} \land\, b = 0 \\ \land\, b' = 1 \end{array}$$

$$Tock \triangleq \begin{array}{l} \land\, b = 1 \\ \land\, b' = 0 \end{array}$$

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 2.8 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*.

We have been viewing the bounded buffer algorithm *BBuf* and its refinements as two-process algorithms. However, they can also quite naturally be viewed as N-process algorithms, the i^{th} process being responsible for reading and writing from buffer element $buf[i]$, for each i in $0\,..\,(N-1)$. Here is algorithm *NProcBBuf*, an N-process version of algorithm *BBuf*. Compare it with algorithm *BBuf*. We show that if we ignore the value of the variable *pc* of algorithm *NProcBBuf*, the two algorithms have the same behaviors. (The TLA$^+$ translation of *NProcBBuf* has a variable *pc*, but the translation of *BBuf* does not.) To state this more precisely, let $Spec_2$ be the TLA$^+$ formula *Spec* in the translation of *BBuf*—the formula that is the specification of algorithm *BBuf*. Let $Spec_N$ be the corresponding formula for algorithm *NProcBBuf*. We show that $Spec_2$ is equivalent to $\exists\, pc : Spec_N$, the formula obtained by hiding the variable□ *pc* in $Spec_N$.

Here is the ASCII text of the algorithm and its enclosing module. First show that *NProcBBuf* implements *BBuf* under the identity refinement mapping—one that defines $\overline{v} = v$ for every variable v of *BBuf*. This shows that every behavior of *NProcBBuf* is a behavior of *BBuf*, so $\exists\, pc : Spec_N$ implies $Spec_2$. We now want show that $Spec_2$ implies $\exists\, pc : Spec_N$. For this, we must show that *BBuf* implements *NProcBBuf* under a refinement mapping that is the identity on all variables of *NProcBBuf* except *pc*. You can do this in:

Question 8.10 Find an invariant of algorithm *NProcBBuf* of the form *pc* = ANSWER

expr that expresses the value of *pc* in terms of the values of the other variables. Use TLC to check that *BBuf* implements *NProcBBuf* under the refinement mapping with $\overline{pc} = expr$ that is the identity on all other variables of *NProcBBuf*.