Implementing and Combining Specifications

Leslie Lamport

02 Sep 2004

Contents

1 Introduction

This note discusses some practical aspects of specification that are not explained clearly or not explained at all in the TLA$^+$ book [2]. I take as an example a very simple lock API that is specified in Section 2. Section 3 discusses the concept of an interface and explains how the specification of the lock API's interface can be put into a separate module. Section 4 gives a simple implementation of the lock API and explains how we show the implementation's correctness. Section 5 explains how to use the lock API's specification in the specification of a system that uses that API.

2 The Example Specification

We now give our example lock API specification, which is formula *LockSpec* of the following module. The specification is simple and is explained by the comments. The rest of this note assumes that you understand the purpose of the declarations and definitions; but you don't have to understand the specification in complete detail.

MODULE *Lock*1

This module specifies a very simple API for a lock. A lock is a resource in a multi-threaded program that can be owned by at most one thread at a time. The API provides two procedures, *Acquire* and *Release*. An *Acquire* call blocks if another thread owns the resource. For simplicity, we avoid the need for error returns by specifying that an *Acquire* call by the lock's current owner returns immediately, and a *Release* call by a thread that doesn't own the resource is a no-op.

CONSTANT *Thread*

The set of all thread identifiers.

$NoThread \triangleq \text{CHOOSE } nt : nt \notin Thread$

An arbitrary element that is not a thread identifier.

$Proc \triangleq \{\text{"acquire"}, \text{"release"}\}$

The set of procedures.

We describe the calling state of the threads by a variable ctl, where $ctl[t]$ is a record that describes the state of thread t. The value of $ctl[t]$ is initially "ready", and it is changed to "waiting" when a procedure is called. There is then an internal step, in which t acquires or releases the lock, that changes $ctl[t]$ to "done". The return from the procedure call resets $ctl[t]$ to "ready". While t is executing a procedure call, $ctl[t].proc$ records which procedure is being called.

In a more typical API, there would be arguments to procedure calls and values returned. These would be recorded in additional components of the record $ctl[t]$. Most API specifications are like this one in that the execution of a procedure call is described with a single internal step (the one that changes $ctl[t]$ from "waiting" to "done"). However, some API specs have procedure calls whose description involves performing more than one internal step. For example, suppose we were specifying a FCFS (first-come, first-served) lock. In that case, an execution of an *Acquire* call when the lock is not free might be described by two internal actions, one that puts the thread onto a waiting queue and another in which the thread acquires the lock. This might be represented by letting $ctl[t]$ have two separate values "waiting1" and "waiting2", or by having both internal actions occur when $ctl[t]$ equals the same value "waiting".

$CtlState \triangleq$

The set of possible values of $ctl[t]$ for a thread t.

$$\begin{array}{ll} & [state : \{\text{"ready"}\}] \\ \cup & [state : \{\text{"waiting"}, \text{"done"}\}, proc : Proc] \end{array}$$

VARIABLES ctl, $owner$

The variable *owner* records the current owner of the lock. It equals *NoThread* if the lock is free.

$TypeInvariant \triangleq$

A predicate describing the type-correct values of *ctl* and *owner*.

$$\begin{array}{l} \land\ ctl \in [Thread \rightarrow CtlState] \\ \land\ owner \in Thread \cup \{NoThread\} \end{array}$$

$Init \triangleq$

The specification's initial condition.

$$\begin{array}{l} \land\ ctl = [t \in Thread \mapsto [state \mapsto \text{"ready"}]] \\ \land\ owner = NoThread \end{array}$$

We now describe the specification's actions—that is, the top-level disjuncts of the next-state action.

$Call(t, proc) \triangleq$

Thread t calls procedure *proc*.

$$\begin{array}{l} \land\ ctl[t].state = \text{"ready"} \\ \land\ ctl' = [ctl \text{ EXCEPT } ![t] = [state \mapsto \text{"waiting"}, proc \mapsto proc]] \\ \land\ \text{UNCHANGED } owner \end{array}$$

$Return(t) \triangleq$

Thread t returns from a procedure call.

$$\begin{aligned} &\land ctl[t].state = \text{"done"} \\ &\land ctl' = [ctl \text{ EXCEPT } ![t] = [state \mapsto \text{"ready"}]] \\ &\land \text{UNCHANGED } owner \end{aligned}$$

$DoAcquire(t) \triangleq$

The internal step in which t acquires ownership of the lock.

$$\begin{aligned} &\land ctl[t].state = \text{"waiting"} \\ &\land ctl[t].proc = \text{"acquire"} \\ &\land owner \in \{NoThread, t\} \\ &\land owner' = t \\ &\land ctl' = [ctl \text{ EXCEPT } ![t].state = \text{"done"}] \end{aligned}$$

$DoRelease(t) \triangleq$

The internal step in which t releases ownership of the lock.

$$\begin{aligned} &\land ctl[t].state = \text{"waiting"} \\ &\land ctl[t].proc = \text{"release"} \\ &\land owner' = \text{IF } owner = t \text{ THEN } NoThread \text{ ELSE } owner \\ &\land ctl' = [ctl \text{ EXCEPT } ![t].state = \text{"done"}] \end{aligned}$$

$Internal(t) \triangleq DoAcquire(t) \lor DoRelease(t)$

For later use, we define $Internal(t)$ to be the disjunction of the internal actions performed by thread t.

$Next \triangleq$

The complete next-state action.

$$\exists t \in Thread : \begin{aligned}[t] &\lor \exists proc \in Proc : Call(t, proc) \\ &\lor Return(t) \\ &\lor Internal(t) \end{aligned}$$

We take as the liveness requirement of the API that every thread eventually returns from any procedure call, unless it is an *Acquire* call and some other thread acquires the lock and never releases it. It is a good exercise in understanding fairness and liveness to convince yourself that this requirement is expressed by conjoining the following fairness condition to the specification.

$Fairness \triangleq \forall t \in Thread : \mathrm{SF}_{\langle ctl, owner \rangle}(Internal(t) \lor Return(t))$

$LockSpec \triangleq$

The complete specification.

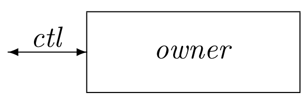
$$\begin{aligned} &\land Init \\ &\land \Box[Next]_{\langle ctl, owner \rangle} \end{aligned}$$

$\land \forall t \in \mathit{Thread} : \mathrm{SF}_{\langle \mathit{ctl},\, \mathit{owner} \rangle}(\mathit{Internal}(t) \lor \mathit{Return}(t))$

THEOREM $\mathit{LockSpec} \Rightarrow \Box \mathit{TypeInvariant}$

3 The Interface

The lock API specification has two variables, *ctl* and *owner*. The variable *owner* records the internal state of the lock; the variable *ctl* describes the call state of the threads using the API. That is, for each thread t, the value of $\mathit{ctl}[t]$ tells if thread t is currently calling the API and, if so, what procedure it is calling and how far it has progressed in executing the call. If we were to draw a picture of the specification, it might look like this:



The picture suggests that the variable *ctl* describes the interaction of the semaphore with its environment, and the variable *owner* describes the internal state of the semaphore. We regard the value of *ctl* to be externally visible, while the value of *owner* cannot be directly observed—it can at best be inferred from observations of the sequence of values assumed by *ctl*.

This picture is actually misleading. If *ctl* were really just an interface variable, its value would be changed only by calls and returns to/from procedures. However, the value of $\mathit{ctl}[t]$ is changed also by the thread t step that occurs between a call and a return. Such a step should really be internal, since there is no way for users of the API to see when that step occurs. (The order in which these steps are performed for two concurrent procedure calls can at best be inferred from the results returned by those calls.) By considering *ctl* to be the interface, we are pretending that there is a “window” into the API that allows its users to see when the step is performed. It would be nicer to let *ctl* model the real interface, so it is changed only by the call and return actions. This would require introducing an additional internal variable to remember when an internal step occurs. However, there is no harm in pretending that the window exists and that users can see when the internal step occurs. Doing so avoids an extra variable, making the spec a little bit simpler. So we will pretend that *ctl* describes the actual interface.

For reasons that will become clear later on, it’s often a good idea to put into a separate *interface module* the part of the specification that describes just the interface. That part of the specification consists of the variable *ctl* and everything that involves it alone. We therefore rewrite module *Lock*1 as two modules, the interface module *LockInterface* and the module *Lock* that extends it and defines the actual API specification, which is formula *LockSpec*. Here is module *LockInterface*

MODULE *LockInterface*

This module declares the parameters and defines the operators that describe just the interface of the lock API specification. The first part of this module consists of the beginning of module *Lock*1, which contain declarations and definitions that pertain to the interface.

CONSTANT *Thread*

$$NoThread \triangleq \text{CHOOSE } nt : nt \notin Thread$$

$$Proc \triangleq \{\text{"acquire"}, \text{"release"}\}$$

$$CtlState \triangleq \begin{array}{ll} & [state : \{\text{"ready"}\}] \\ \cup & [state : \{\text{"waiting"}, \text{"done"}\}, proc : Proc] \end{array}$$

VARIABLE *ctl*

For future use, we give names to some conjuncts in the definitions from module Lock1 that mention the interface variable *ctl*. First, we define *IntTypeInvariant* and *IntInit* to be the conjuncts of *TypeInvariant* and *Init* that describe the interface variable *ctl*.

$$IntTypeInvariant \triangleq ctl \in [Thread \rightarrow CtlState]$$
$$IntInit \triangleq ctl = [t \in Thread \mapsto [state \mapsto \text{"ready"}]]$$

We now give names to the conjuncts of the actions $Call(t, proc)$ and $Return(t)$ from module *Lock*1 that mention the interface variable *ctl*.

$$IntCall(t, proc) \triangleq \begin{array}{l} \land\ ctl[t].state = \text{"ready"} \\ \land\ ctl' = [ctl \text{ EXCEPT } ![t] = [state \mapsto \text{"waiting"}, proc \mapsto proc]] \end{array}$$

$$IntReturn(t) \triangleq \begin{array}{l} \land\ ctl[t].state = \text{"done"} \\ \land\ ctl' = [ctl \text{ EXCEPT } ![t] = [state \mapsto \text{"ready"}]] \end{array}$$

Module *Lock* is obtained in a straightforward manner from the part of module *Lock*1 not subsumed by the *LockInterface* module. It begins:

MODULE *Lock*

EXTENDS *LockInterface*

VARIABLE *owner*

The rest of module *Lock* is the same as the corresponding part of module *Lock*1, except for the definitions of *Init*, *TypeInvariant*, *Call*, and *Return*. In those definitions, conjuncts that were given names in module *LockInterface* are replaced by those names. For example, the definition of *Init* becomes

$$
\begin{aligned}
&\mathit{Init} \ \triangleq \\
&\quad \land\ \mathit{IntInit} \\
&\quad \land\ \mathit{owner} = \mathit{NoThread}
\end{aligned}
$$

and the definition of $\mathit{Return}(t)$ becomes

$$
\begin{aligned}
&\mathit{Return}(t) \ \triangleq \\
&\quad \land\ \mathit{IntReturn}(t) \\
&\quad \land\ \textsc{unchanged}\ \mathit{owner}
\end{aligned}
$$

Formula *LockSpec*, the specification of the lock API, describes the allowed behaviors (sequences of values) of variables *ctl* and *owner*. But a user of the lock API interacts with the API by using the interface. It sees only the *ctl* variable, not the *owner* variable. It cares only about the behavior of *ctl*, not of *owner*. A philosophically correct spec of the API would say that *ctl* behaves as if there were a variable *owner* such that the behaviors of *ctl* and *owner* satisfy formula *LockSpec*. Such a specification is written informally as

$$\boldsymbol{\exists}\, \mathit{owner} : \mathit{LockSpec}$$

For good reasons that do not concern us here, we can't write the specification in that way. Instead, the philosophically correct specification *PCLockSpec* is defined in the following module *PCLock*.

MODULE *PCLock*

EXTENDS *LockInterface*

$\mathit{Inner}(\mathit{owner}) \triangleq$ INSTANCE *Lock*
$\mathit{PCLockSpec} \triangleq \boldsymbol{\exists}\, \mathit{owner} : \mathit{Inner}(\mathit{owner})!\mathit{LockSpec}$

However, we will pretend that *PCLockSpec* is defined by

$$\mathit{PCLockSpec} \ \triangleq\ \boldsymbol{\exists}\, \mathit{owner} : \mathit{LockSpec}$$

4 Implementation

We now implement the lock API using a trivial version of the bakery algorithm [1]. The bakery algorithm works roughly as follows. A thread that wants to acquire lock chooses a number, and the thread with the smallest number gains ownership of the lock. We let $num[t]$ be the number of thread t, which initially equals 0. A thread t that wants to acquire the lock sets $num[t]$ to 1 plus the largest value of $num[u]$ for every thread u. Our trivialized version of the algorithm has thread t perform the entire operation of reading the values of all the $num[u]$ and setting $num[t]$ as a single atomic step.

We begin by describing the algorithm in vaguely C-like pseudo-code. The program for thread t is as follows, where initially $num[t] = 0$. Statements are given labels for later use.

```
Acquire
  a: if (num[t] == 0) {
       num[t] = 1 + largest num[u] for all threads u}
     else {
       goto c} ;
  b: wait until
      (for all threads u : (num[u] = 0) or (num[u] >= num[t]);
  c: return ;

Release
  a: num[t] = 0 ;
  c: return ;
```

The TLA$^+$ specification of this algorithm is formula *BakerySpec* of the following module *BakeryLock*. There is an obvious correspondence between the specification's actions and the statements in the pseudo-code above, except that the specification has a procedure-call action that does not appear explicitly in the pseudo-code.

MODULE *BakeryLock*

EXTENDS *LockInterface*, *Naturals*

$Max(S) \triangleq$ CHOOSE $n \in S : \forall\, m \in S : n \geq m$

If S is a non-empty set of numbers, then $Max(S)$ is its maximum element.

VARIABLES pc, num

$pc[t]$ equals the "program counter" of thread t—that is, the label of the next statement to be executed by t. When thread t is not executing a procedure call, $pc[t]$ equals "a".

$$
\begin{array}{l}
TypeInvariant \triangleq \quad \text{The type invariant predicate} \\
\quad \land\ IntTypeInvariant \\
\quad \land\ pc \quad\ \in [Thread \rightarrow \{\text{"a"}, \text{"b"}, \text{"c"}\}] \\
\quad \land\ num \in [Thread \rightarrow Nat]
\end{array}
$$

$$
\begin{array}{l}
Init \triangleq \quad \text{The initial predicate} \\
\quad \land\ IntInit \\
\quad \land\ pc \quad\ = [t \in Thread \mapsto \text{"a"}] \\
\quad \land\ num = [t \in Thread \mapsto 0]
\end{array}
$$

$GoTo(t, c1, c2) \triangleq$

An action expression asserting that control in thread t moves from label $c1$ to label $c2$.

$$
\begin{array}{l}
\quad \land\ pc[t] = c1 \\
\quad \land\ pc' = [pc \text{ EXCEPT } ![t] = c2]
\end{array}
$$

Below are the algorithm's actions—that is, the disjuncts of the next-state action. The value of $ctl[t].state$ changes from "waiting" to "done" when thread t reaches the *return* statement (labeled c) in either procedure. Executing that *return* statement changes $ctl[t].state$ to "ready".

$Call(t, proc) \triangleq$

The action of thread t calling procedure *proc*.

$$
\begin{array}{l}
\quad \land\ IntCall(t, proc) \\
\quad \land\ \text{UNCHANGED } \langle pc, num \rangle
\end{array}
$$

$AcqStepA(t) \triangleq$

The action of thread t executing statement a of the *Acquire* procedure. The first two conjuncts assert that t is inside a call of *Acquire*. Because of the first conjuncts of the THEN and ELSE parts of the IF formula, the action is enabled only if $ctl[t] =$ "a". In that case, the IF condition is false ($num[t]$ is greater than 0) iff thread t already owns the lock.

$$
\begin{array}{l}
\quad \land\ ctl[t].state \neq \text{"ready"} \\
\quad \land\ ctl[t].proc \ = \text{"acquire"} \\
\quad \land\ \text{IF } num[t] = 0 \\
\qquad\quad \text{THEN } \land\ GoTo(t, \text{"a"}, \text{"b"}) \\
\qquad\qquad\qquad \land\ num' = [num \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad\qquad ![t] = \text{IF } @ = 0 \text{ THEN } 1 + Max(\{num[u] : u \in Thread\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } @] \\
\qquad\qquad\qquad \land\ \text{UNCHANGED } ctl \\
\qquad\quad \text{ELSE } \ \land\ GoTo(t, \text{"a"}, \text{"c"}) \\
\qquad\qquad\qquad \land\ ctl' = [ctl \text{ EXCEPT } ![t].state = \text{"done"}] \\
\qquad\qquad\qquad \land\ \text{UNCHANGED } num
\end{array}
$$

$AcqStepB(t) \triangleq$

The action of thread t executing statement b of the *Acquire* procedure. The first conjunct implies that $pc[t]$ equals "b", which is possible only if t is calling *Acquire*.

$$\begin{array}{l} \land\ GoTo(t, \text{"b"}, \text{"c"}) \\ \land\ \forall u \in Thread : (num[u] = 0) \lor (num[u] \geq num[t]) \\ \land\ ctl' = [ctl \text{ EXCEPT } ![t].state = \text{"done"}] \\ \land\ \text{UNCHANGED } num \end{array}$$

$Return(t) \triangleq$

The action of thread t executing the *return* step (which has label c) in either the *Acquire* or *Release* procedure.

$$\begin{array}{l} \land\ GoTo(t, \text{"c"}, \text{"a"}) \\ \land\ IntReturn(t) \\ \land\ \text{UNCHANGED } num \end{array}$$

$RelStepA(t) \triangleq$

The action of thread t executing statement a of the *Release* procedure. The first two conjuncts assert that t is inside a call of *Release*.

$$\begin{array}{l} \land\ ctl[t].state \neq \text{"ready"} \\ \land\ ctl[t].proc\ = \text{"release"} \\ \land\ GoTo(t, \text{"a"}, \text{"c"}) \\ \land\ num' = [num \text{ EXCEPT } ![t] = 0] \\ \land\ ctl' \quad = [ctl \text{ EXCEPT } ![t].state = \text{"done"}] \end{array}$$

$ImplAction(t) \triangleq$

For convenience, we define $ImplAction(t)$ to be the disjunction of all of thread t's actions that are performed by the lock implementation itself—which means every action except the procedure call, which is performed by the user of the API.

$$\begin{array}{l} \lor\ AcqStepA(t) \\ \lor\ AcqStepB(t) \\ \lor\ Return(t) \\ \lor\ RelStepA(t) \end{array}$$

$Next \triangleq$

The next-state action.

$$\exists t \in Thread : \begin{array}{l} \lor\ \exists proc \in Proc : Call(t, proc) \\ \lor\ ImplAction(t) \end{array}$$

$BakerySpec \triangleq$

The complete specification. The liveness condition is weak fairness on the implementation action of every thread.

$$\begin{array}{l} \land\ Init \\ \land\ \Box[Next]_{\langle ctl, pc, num \rangle} \end{array}$$

$\land \forall t \in \mathit{Thread} : \mathrm{WF}_{\langle ctl,\, pc,\, num \rangle}(\mathit{ImplAction}(t))$

THEOREM $\mathit{BakerySpec} \Rightarrow \Box \mathit{TypeInvariant}$

I claim that the algorithm specified by formula *BakerySpec* implements the lock API. This means that *BakerySpec* implies the specification *PCLockSpec* of the lock API—the specification with *owner* hidden. In other words, the formula

$$\mathit{BakerySpec} \Rightarrow \mathit{PCLock}$$

should be valid. This can be asserted in module *BakeryLock* as follows:

$\mathit{Spec} \triangleq$ INSTANCE PCLock
THEOREM $\mathit{BakerySpec} \Rightarrow \mathit{Spec}!\mathit{PCLock}$

Instead of instantiating module *PCLock*, module *BakeryLock* could extend it—assuming there were no name conflicts. But instantiating it with renaming in this way works even if there are name conflicts.

Unfortunately, TLC cannot check the correctness of this theorem. Remembering the definition of *PCLockSpec*, this theorem is equivalent to

$$\mathit{BakerySpec} \Rightarrow \boldsymbol{\exists}\, \mathit{owner} : \mathit{LockSpec}$$

and TLC does not handle the hiding operator $\boldsymbol{\exists}$. To figure out how to get TLC to check correctness of the implementation, we must examine what it means for *BakerySpec* to imply (or implement) *PCLockSpec*.

Let the set *Threads* of threads be $\{t1,\, t2\}$, and let the operators :> and @@ be defined as in the standard *TLC* module so that

$$f \triangleq (t1 :> 3 \;@@\; t2 :> 17)$$

defines f to be the function with domain $\{t1,\, t2\}$ such that $f[t1] = 3$ and $f[t2] = 17$.

Spec *BakerySpec* implements/implies spec *PCLockSpec* iff every behavior that satisfies *BakerySpec* also satisfies *PCLockSpec*. Suppose someone gives us the following behavior that satisfies *BakerySpec*

<u>Behavior 1</u>

$$\left[\begin{array}{ll} ctl & = (t1 :> [state \mapsto \text{“ready”}] \;@@ \\ & \quad\;\; t2 :> [state \mapsto \text{“ready”}]) \\ num & = (t1 :> 0 \;@@\; t2 :> 0) \\ pc & = (t1 :> \text{“a”} \;@@\; t2 :> \text{“a”}) \end{array}\right]$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{"waiting"},\ proc \mapsto \text{"acquire"}]\ @@ \\ & \quad t2 :> [state \mapsto \text{"ready"}]) \\ num & = (t1 :> 0\ @@\ t2 :> 0) \\ pc & = (t1 :> \text{"a"}\ @@\ t2 :> \text{"a"}) \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{"waiting"},\ proc \mapsto \text{"acquire"}]\ @@ \\ & \quad t2 :> [state \mapsto \text{"ready"}]) \\ num & = (t1 :> 1\ @@\ t2 :> 0) \\ pc & = (t1 :> \text{"b"}\ @@\ t2 :> \text{"a"}) \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{"done"},\ proc \mapsto \text{"acquire"}]\ @@ \\ & \quad t2 :> [state \mapsto \text{"ready"}]) \\ num & = (t1 :> 1\ @@\ t2 :> 0) \\ pc & = (t1 :> \text{"c"}\ @@\ t2 :> \text{"a"}) \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{"ready"}]\ @@ \\ & \quad t2 :> [state \mapsto \text{"ready"}]) \\ num & = (t1 :> 1\ @@\ t2 :> 0) \\ pc & = (t1 :> \text{"a"}\ @@\ t2 :> \text{"a"}) \end{bmatrix}$$

$$\downarrow$$

$$\ldots$$

We must show that this behavior satisfies *PCLockSpec*. How do we do that?

Let's examine what it means for a behavior to satisfy *PCLockSpec*. The only free variable that appears in *PCLockSpec* is *ctl*. So, whether or not a behavior satisfies *PCLockSpec* depends only on the values that *ctl* assumes in that behavior. So, we can forget about the values of *mem* and *pc* and ask if the following behavior, obtained from Behavior 1 by deleting the values of *num* and *pc*, satisfies *PCLockSpec*.

<u>Behavior 2</u>

$$\begin{bmatrix} ctl = (t1 :> [state \mapsto \text{"ready"}]\ @@ \\ \quad t2 :> [state \mapsto \text{"ready"}]) \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl = (t1 :> [state \mapsto \text{"waiting"},\ proc \mapsto \text{"acquire"}]\ @@ \\ \quad t2 :> [state \mapsto \text{"ready"}]) \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl = (t1 :> [state \mapsto \text{“waiting”},\ proc \mapsto \text{“acquire”}]\ @@ \\ \qquad\quad t2 :> [state \mapsto \text{“ready”}]) \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl = (t1 :> [state \mapsto \text{“done”},\ proc \mapsto \text{“acquire”}]\ @@ \\ \qquad\quad t2 :> [state \mapsto \text{“ready”}]) \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl = (t1 :> [state \mapsto \text{“ready”}]\ @@ \\ \qquad\quad t2 :> [state \mapsto \text{“ready”}]) \end{bmatrix}$$

$$\downarrow$$

$$\ldots$$

By definition, Behavior 2 satisfies *PCLockSpec* iff we can invent some values for the variable *owner* to produce a behavior satisfying *LockSpec*. We can do that as follows.

<u>Behavior 3</u>

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{“ready”}]\ @@ \\ & \qquad t2 :> [state \mapsto \text{“ready”}]) \\ owner & = NoThread \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{“waiting”},\ proc \mapsto \text{“acquire”}]\ @@ \\ & \qquad t2 :> [state \mapsto \text{“ready”}]) \\ owner & = NoThread \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{“waiting”},\ proc \mapsto \text{“acquire”}]\ @@ \\ & \qquad t2 :> [state \mapsto \text{“ready”}]) \\ owner & = NoThread \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{“done”},\ proc \mapsto \text{“acquire”}]\ @@ \\ & \qquad t2 :> [state \mapsto \text{“ready”}]) \\ owner & = t1 \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{"ready"}] \;@@ \\ & \quad\; t2 :> [state \mapsto \text{"ready"}]) \\ owner & = t1 \end{bmatrix}$$

$$\downarrow$$

$$\ldots$$

It's easy to check that this behavior satisfies formula *LockSpec*. (Remember that all TLA$^+$ specifications allow stuttering steps, so *LockSpec* allows steps that leave *ctl* and *owner* unchanged.) Therefore Behavior 1, which satisfies *BakerySpec*, satisfies *PCLockSpec*.

To show that *BakerySpec* implies *PCLockSpec*, we have to show that every behavior satisfying *BakerySpec* also satisfies *PCLockSpec*. Suppose we could define an expression $\overline{owner}$ in terms of the variables *ctl*, *num*, and *pc* such that the value of $\overline{owner}$ in each state of a behavior provided the values of *owner* needed to show that the behavior satisfies *LockSpec*. For example, the value of $\overline{owner}$ in each of the states of Behavior 1 would be as follows:

<u>Behavior 1</u>

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{"ready"}] \;@@ \\ & \quad\; t2 :> [state \mapsto \text{"ready"}]) \\ num & = (t1 :> 0 \;@@\; t2 :> 0) \\ pc & = (t1 :> \text{"a"} \;@@\; t2 :> \text{"a"}) \end{bmatrix}$$

$$\overline{owner} = NoThread$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{"waiting"},\ proc \mapsto \text{"acquire"}] \;@@ \\ & \quad\; t2 :> [state \mapsto \text{"ready"}]) \\ num & = (t1 :> 0 \;@@\; t2 :> 0) \\ pc & = (t1 :> \text{"a"} \;@@\; t2 :> \text{"a"}) \end{bmatrix}$$

$$\overline{owner} = NoThread$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{"waiting"},\ proc \mapsto \text{"acquire"}] \;@@ \\ & \quad\; t2 :> [state \mapsto \text{"ready"}]) \\ num & = (t1 :> 1 \;@@\; t2 :> 0) \\ pc & = (t1 :> \text{"b"} \;@@\; t2 :> \text{"a"}) \end{bmatrix}$$

$$\overline{owner} = NoThread$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{"done"},\ proc \mapsto \text{"acquire"}]\ @@ \\ & \quad\ \ t2 :> [state \mapsto \text{"ready"}]) \\ num & = (t1 :> 1\ @@\ t2 :> 0) \\ pc & = (t1 :> \text{"c"}\ @@\ t2 :> \text{"a"}) \end{bmatrix}$$

$$\overline{owner} = t1$$

$$\downarrow$$

$$\begin{bmatrix} ctl & = (t1 :> [state \mapsto \text{"ready"}]\ @@ \\ & \quad\ \ t2 :> [state \mapsto \text{"ready"}]) \\ num & = (t1 :> 1\ @@\ t2 :> 0) \\ pc & = (t1 :> \text{"a"}\ @@\ t2 :> \text{"a"}) \end{bmatrix}$$

$$\overline{owner} = t1$$

$$\downarrow$$

$$\ldots$$

In other words, if in Behavior 1 we let the value of the variable *owner* always equal the value of the expression $\overline{owner}$, we get a behavior that satisfies *LockSpec*. An equivalent way of saying this is that Behavior 1 satisfies the formula $\overline{LockSpec}$ obtained by substituting the expression $\overline{owner}$ for the variable *owner*. If this is true not just of Behavior 1, but of every behavior satisfying *BakerySpec*, then *BakerySpec* implies *PCLockSpec*.

We can therefore prove that *BakerySpec* implies *PCLockSpec* by finding an expression $\overline{owner}$ such that every behavior satisfying *BakerySpec* satisfies $\overline{LockSpec}$. But every behavior satisfying *BakerySpec* satisfies $\overline{LockSpec}$ iff *BakerySpec* implies $\overline{LockSpec}$. Thus, to prove that *BakerySpec* implies *PCLockSpec*, it suffices to find an expression $\overline{owner}$ such that $BakerySpec \Rightarrow \overline{LockSpec}$ is a valid theorem, where $\overline{LockSpec}$ is the formula obtained by substituting $\overline{owner}$ for *owner* in *LockSpec*.

To find a suitable expression $\overline{owner}$, we observe that in our simplified bakery algorithm, a thread t owns the lock iff $num[t] > 0$ and t is not waiting at statement b of the *Acquire* procedure. In terms of our TLA$^+$ specification, this means that t owns the lock iff the state predicate $IsOwner(t)$, defined as follows, is true.

$$\begin{aligned} IsOwner(t) \ \triangleq\ & \land\ num[t] \neq 0 \\ & \land\ \neg \land\ ctl[t].state = \text{"waiting"} \\ & \qquad \land\ ctl[t].proc\ = \text{"acquire"} \\ & \qquad \land\ pc[t] = \text{"b"} \end{aligned}$$

We can then define $\overline{owner}$ to equal

$$
\begin{array}{l}
\textsc{if}\ \exists\, t \in \mathit{Thread} : \mathit{IsOwner}(t) \\
\quad \textsc{then}\ \textsc{choose}\ t \in \mathit{Thread} : \mathit{IsOwner}(t) \\
\quad \textsc{else}\ \ \mathit{NoThread}
\end{array}
$$

This is all expressed in TLA$^+$ in the following module, where $\overline{owner}$ is written $\mathit{ownerBar}$, and $\overline{\mathit{LockSpec}}$ becomes $\mathit{Bar}!\mathit{LockSpec}$.

MODULE *BakeryLockCorrect*

EXTENDS *BakeryLock*

$$
\begin{array}{l}
\mathit{ownerBar} \triangleq \qquad \text{We write } \mathit{ownerBar} \text{ instead of } \overline{owner} \\
\quad \textsc{let}\ \mathit{IsOwner}(t) \triangleq \ \land\ num[t] \neq 0 \\
\qquad\qquad\qquad\qquad\quad\ \land\ \lnot \land\ ctl[t].state = \text{“waiting”} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \land\ ctl[t].proc = \text{“acquire”} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \land\ pc[t] = \text{“b”} \\
\quad \textsc{in}\quad \textsc{if}\ \exists\, t \in \mathit{Thread} : \mathit{IsOwner}(t) \\
\qquad\qquad \textsc{then}\ \textsc{choose}\ t \in \mathit{Thread} : \mathit{IsOwner}(t) \\
\qquad\qquad \textsc{else}\ \ \mathit{NoThread}
\end{array}
$$

$\mathit{Bar} \triangleq$ INSTANCE *Lock* WITH $owner \leftarrow \mathit{ownerBar}$

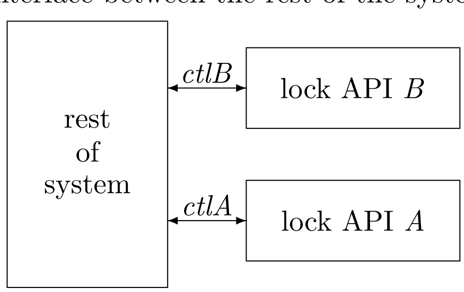
For every operator or formula F defined in module *Lock*, this statement causes $\mathit{Bar}!F$ to be defined to equal the operator or formula obtained from F by substituting $\mathit{ownerBar}$ for $owner$. Hence, $\mathit{Bar}!\mathit{LockSpec}$ equals $\overline{\mathit{LockSpec}}$

THEOREM $\mathit{BakerySpec} \Rightarrow \mathit{Bar}!\mathit{LockSpec}$

TLC can check this theorem.

5 Combining Specifications

Suppose we are writing a specification *SysSpec* of some system that uses one or more locks. For concreteness, suppose it uses two locks A and B. We can picture the system as follows, where *ctlA* and *ctlB* are variables representing the interface between the rest of the system and the two lock APIs.



We expect the two lock APIs to be described by our lock API specification, so we expect two "copies" of that specification to appear in the definition of *SysSpec*, one with *ctlA* substituted for *ctl* and the other with *ctlB* substituted for *ctl*.

The philosophically correct specification of a single lock API is formula *PCLockSpec* of module *PCLock*. As explained in Chapter 10 of the TLA$^+$ book, the philosophically correct specification of the system would therefore have the form

$$RestOfSys \land PCLockSpecA \land PCLockSpecB$$

where *PCLockSpecA* and *PCLockSpecB* are formulas *PCLockSpec* with *ctl* instantiated by *ctlA* and *ctlB*, respectively, and *RestOfSys* specifies the rest of the system. More precisely, formula *PCLockSpecA* would be defined by

$$PCLockA \triangleq \textsc{instance}\ PCLock\ \textsc{with}\ ctl \leftarrow ctlA$$
$$PCLockSpecA \triangleq PCLockA!PCLockSpec$$

and *PCLockSpecB* would be defined analogously.

There are two problems with this philosophically correct approach. The first is that writing *RestOfSys* is somewhat tricky. The second is that TLC can't handle this kind of "compositional" specification. So, we'll do it an easier way, using two copies of module *Lock* instead of module *PCLock*.

As an example, let's suppose we are modeling a multi-threaded program that uses two locks, which we call locks A and B. Suppose the program for each thread contains the following piece of code which might arise if x and y are shared variables protected by locks A and B, respectively:

```
    ...
a : Acquire Lock A
b : Acquire Lock B
c : x = x + y
d : Release Lock B
e : Release Lock A
    ...
```

To model this program in TLA$^+$, we use variables x and y to represent the program variables `x` and `y`, and represent program control with a variable *pc*—as we did above in module *BakeryLock*. Letting *Thread* denote the set of threads, we might begin the specification as follows. (The ... represents the other specification variables, including ones representing other program variables.)

$\text{EXTENDS } \mathit{Naturals}$
$\text{CONSTANT } \mathit{Thread}$
$\text{VARIABLES } \mathit{pc},\ x,\ y,\ \ldots$

We would "import" two copies of the *Lock* module, with different instantiations of that module's variables. We represent lock A with a copy in which *ctlA* is substituted for *ctl* and *ownerA* is substituted for *owner*; and similarly for lock B.

$$
\begin{array}{l}
\text{VARIABLES } \mathit{ctlA},\ \mathit{ownerA},\ \mathit{ctlB},\ \mathit{ownerB} \\
\mathit{LockA} \triangleq \text{INSTANCE } \mathit{Lock} \text{ WITH } \mathit{ctl} \leftarrow \mathit{ctlA},\ \mathit{owner} \leftarrow \mathit{ownerA} \\
\mathit{LockB} \triangleq \text{INSTANCE } \mathit{Lock} \text{ WITH } \mathit{ctl} \leftarrow \mathit{ctlB},\ \mathit{owner} \leftarrow \mathit{ownerB}
\end{array}
$$

The other parameter of module *Lock*, the constant *Thread*, is instantiated by the constant *Thread* of the current module. For convenience, we make the following definitions:

$$
\begin{array}{l}
\mathit{aVars} \triangleq \langle \mathit{ctlA},\ \mathit{ownerA} \rangle \\
\mathit{bVars} \triangleq \langle \mathit{ctlB},\ \mathit{ownerB} \rangle
\end{array}
$$

Our specification uses definitions from the two instances of the *Lock* module in the initial predicate, the type invariant, and the next-state action. The initial predicate is:

$$
\begin{array}{lcl}
\mathit{Init} & \triangleq & \land\ \mathit{LockA}!\mathit{Init} \\
 & & \land\ \mathit{LockB}!\mathit{Init} \\
 & & \land\ \mathit{pc} = [t \in \mathit{Thread} \mapsto \ \ldots] \\
 & & \land\ x = \ \ldots \\
 & & \land\ y = \ \ldots \\
 & & \ldots
\end{array}
$$

The first two conjuncts specify the initial values of the variables *ctlA*, *ownerA*, *ctlB*, and *ownerB*. Similarly, the type invariant is:

$$
\begin{array}{lcl}
\mathit{TypeInvariant} & \triangleq & \land\ \mathit{LockA}!\mathit{TypeInvariant} \\
 & & \land\ \mathit{LockB}!\mathit{TypeInvariant} \\
 & & \land\ \mathit{pc} \in \ \ldots \\
 & & \ldots
\end{array}
$$

We now examine how we represent the program statements that acquire and release the locks—for example, the statement:

```
a : Acquire Lock A
```

In our model, an execution of this statement by a thread t consists of three steps: the call of the *acquire* procedure, the internal step of the lock in which control changes from "waiting" to "done", and the return. The call and return are described by the following actions, where again "..." stands for the additional variables of the specification:

$$
\begin{array}{l}
\mathit{StepACall}(t) \;\triangleq \\
\quad \land\ pc[t] = \text{"a"} \\
\quad \land\ \mathit{LockA}!\mathit{Call}(t,\ \text{"acquire"}) \\
\quad \land\ \text{UNCHANGED}\ \langle pc,\ x,\ y,\ \ldots,\ \mathit{bVars}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{StepAReturn}(t) \;\triangleq \\
\quad \land\ pc[t] = \text{"a"} \\
\quad \land\ \mathit{LockA}!\mathit{Return}(t) \\
\quad \land\ pc' = [pc\ \text{EXCEPT}\ ![t] = \text{"b"}] \\
\quad \land\ \text{UNCHANGED}\ \langle x,\ y,\ \ldots,\ \mathit{bVars}\rangle
\end{array}
$$

Note that we must specify that the variables x, y, ..., *ctlB*, and *ownerB* are left unchanged. (Leaving *varsB* unchanged is equivalent to leaving both *ctlB* and *ownerB* unchanged.) The internal steps for all of thread t's lock A procedure calls will be described later by a single action.

The acquiring of lock B by program statement `b` is handled similarly. Execution of the statement

```
c : x = x + y
```

would probably be modeled as a single step satisfying the action

$$
\begin{array}{l}
\mathit{StepC}(t) \;\triangleq \\
\quad \land\quad pc[t] = \text{"c"} \\
\quad \land\quad x' = x + y \\
\quad \land\quad pc' = [pc\ \text{EXCEPT}\ ![t] = \text{"d"}] \\
\quad \land\quad \text{UNCHANGED}\ \langle y,\ \ldots,\ \mathit{aVars},\ \mathit{bVars}\rangle
\end{array}
$$

The statements `d` and `e` would be represented similarly. For example, statement `d` would be described by the two actions

$$
\begin{array}{l}
\mathit{StepDCall}(t) \;\triangleq \\
\quad \land\ pc[t] = \text{"d"} \\
\quad \land\ \mathit{LockB}!\mathit{Call}(t,\ \text{"release"}) \\
\quad \land\ \text{UNCHANGED}\ \langle pc,\ x,\ y,\ \ldots,\ \mathit{aVars}\rangle
\end{array}
$$

$$
\mathit{StepDReturn}(t) \;\triangleq
$$

$$
\begin{array}{l}
\land\ pc[t] = \text{"d"} \\
\land\ LockB!Return(t) \\
\land\ pc' = [pc \text{ EXCEPT } ![t] = \text{"e"}] \\
\land\ \text{UNCHANGED } \langle x,\ y,\ \ldots,\ aVars \rangle
\end{array}
$$

The next-state action would be as follows, where the last two disjuncts describe the internal steps performed by all of thread t's calls to the lock procedures.

$$
\begin{array}{l}
Next \triangleq \\
\quad \exists\, t \in Thread : \begin{array}[t]{l}
\lor\ StepACall(t) \\
\lor\ StepAReturn(t) \\
\ldots \\
\lor\ \land\ LockA!Internal(t) \\
\quad\ \land\ \text{UNCHANGED } \langle pc,\ x,\ y,\ \ldots,\ bVars \rangle \\
\lor\ \land\ LockB!Internal(t) \\
\quad\ \land\ \text{UNCHANGED } \langle pc,\ x,\ y,\ \ldots,\ aVars \rangle
\end{array}
\end{array}
$$

Those two disjuncts would probably also appear in any fairness requirements.

TLC can handle this specification. However, getting it to do so requires one non-obvious trick. Recall that module *LockInterface* has the definition

$$NoThread \triangleq \text{CHOOSE } nt : nt \notin Thread$$

which TLC cannot handle. Hence, one must tell TLC to assign a model value to the constant *NoThread*. (We usually assign the model value `NoThread` to this constant.) Such an assignment is usually performed by putting the statement

```
NoThread  =  NoThread
```

into the `CONSTANTS` part of the configuration file. However, that doesn't work here because *NoThread* is defined in an instantiated (rather than an extended) module. Instead, you can use the following statement in the configuration file:

```
NoThread  =[LockInterface]  NoThread
```

which tells TLC to make the assignment in the *LockInterface* module. Equivalently, you can instead tell TLC to make the assignment in the *Lock* module with the assignment

```
NoThread  =[Lock]  NoThread
```

6 A Simpler Lock Specification

Consider again the example multi-threaded program of Section 5, which contained the following piece of code.

```
    ...
a : Acquire Lock A
b : Acquire Lock B
c : x = x + y
d : Release Lock B
e : Release Lock A
    ...
```

We represented each of the `Acquire` and `Release` statements by two actions. In addition, the next-state action contained two disjuncts describing the internal steps of each of the locks. In many cases, we would like to model an execution of `Acquire` or `Release` as a single step. In addition to simplifying the specification, reducing the execution of each procedure call from three steps to one step would decrease the size of the state space, making model checking easier.

We could obtain this kind of simple model of the program by starting with a simpler model of a lock. The following module specifies such a simpler model; it is simple enough that it should require no explanation.

MODULE *SimpleLock*

CONSTANT *Thread*

$NoThread \triangleq \text{CHOOSE } nt : nt \notin Thread$

VARIABLE *owner*

$Init \triangleq owner = NoThread$

$TypeInvariant \triangleq owner \in Thread \cup \{NoThread\}$

$$Acquire(t) \triangleq \begin{array}{l} \land\ owner \in \{NoThread,\ t\} \\ \land\ owner' = t \end{array}$$

$Release(t) \triangleq \land\ owner' = \text{IF } owner = t \text{ THEN } NoThread \text{ ELSE } owner$

$Next \triangleq \exists\, t \in Thread : Acquire(t) \lor Release(t)$

$SimpleLockSpec \triangleq Init \land \Box[Next]_{owner}$

The first part of our specification of the program that uses the two locks A and B would be just like the one in Section 5:

$$
\begin{array}{l}
\textsc{extends}\ \mathit{Naturals} \\
\textsc{constant}\ \mathit{Thread} \\
\textsc{variables}\ \mathit{pc},\ x,\ y,\ \ldots
\end{array}
$$

$$
\begin{array}{l}
\textsc{variables}\ \mathit{ctlA},\ \mathit{ownerA},\ \mathit{ctlB},\ \mathit{ownerB} \\
\mathit{LockA} \triangleq \textsc{instance}\ \mathit{SimpleLock}\ \textsc{with}\ \mathit{ctl} \leftarrow \mathit{ctlA},\ \mathit{owner} \leftarrow \mathit{ownerA} \\
\mathit{LockB} \triangleq \textsc{instance}\ \mathit{SimpleLock}\ \textsc{with}\ \mathit{ctl} \leftarrow \mathit{ctlB},\ \mathit{owner} \leftarrow \mathit{ownerB}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Init} & \triangleq & \land\ \mathit{LockA}!\mathit{Init} \\
 & & \land\ \mathit{LockB}!\mathit{Init} \\
 & & \ldots
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{TypeInvariant} & \triangleq & \land\ \mathit{LockA}!\mathit{TypeInvariant} \\
 & & \land\ \mathit{LockB}!\mathit{TypeInvariant} \\
 & & \ldots
\end{array}
$$

However, the descriptions of the program statements would be simpler. For example, statement `a` would be described by the single action

$$
\begin{array}{l}
\mathit{StepA}(t) \triangleq \\
\quad \land\ \ \mathit{pc}[t] = \text{“}\mathsf{a}\text{”} \\
\quad \land\ \ \mathit{LockA}!\mathit{Acquire}(t) \\
\quad \land\ \ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![t] = \text{“}\mathsf{b}\text{”}] \\
\quad \land\ \ \textsc{unchanged}\ \langle x,\ y,\ \ldots,\ \mathit{ownerB} \rangle
\end{array}
$$

There would be no extra disjuncts of the next-state relation, since the simple lock specification has no extra internal steps.

Which is the best specification of a lock, the one in module *Lock* or the simple one of module *SimpleLock*? Neither. There is no “best” specification of any system. A specification is written for a purpose. What kind of specification you write depends on that purpose. Module *Lock* was written to specify an API. Module *SimpleLock* doesn't provide much of a specification of a lock API. In fact, formula *Spec* of module *SimpleLock* is equivalent to the formula

$$
\begin{array}{l}
\land\ \mathit{owner} = \mathit{NoThread} \\
\land\ \Box\ [\ \lor\ \land\ \mathit{owner} = \mathit{NoThread} \\
\qquad\qquad \land\ \mathit{owner}' \in \mathit{Thread} \\
\qquad\ \ \lor\ \land\ \mathit{owner} \in \mathit{Thread} \\
\qquad\qquad \land\ \mathit{owner}' = \mathit{NoThread} \\
\quad\ \ ]_{\mathit{owner}}
\end{array}
$$

Both formulas describe the same possible sequences of changes to the variable *owner*.[1] This equivalent way of writing it shows that the specification of module *SimpleLock* isn't a very good one for explaining how to use a lock.

If one writes two different specifications of the same system for different purposes, there will most likely not be any simple formal relation between those two specifications. In this particular example, specification *LockSpec* of module *Lock* implements specification *SimpleLockSpec* of module *SimpleLock*. (Such an implementation relation will hold whenever we simplify an API specification by eliminating the calls and returns and just leaving the internal steps.) However, the two specifications of the program that uses the locks would be different. For example, in the one we described in Section 5, the value of *ownerA* and $pc[t]$ change in separate steps during the execution of statement `a`; in the specification sketched in this section, the $StepA(t)$ action changes *ownerA* and $pc[t]$ in a single step.

Although it does not yet happen often in industrial applications, you may sometimes want a specification to serve two functions. In our toy example, we wanted to use the lock API's specification both to check an implementation of the API and to check a program that uses the API. It's nice to check the program using the same specification of the API that we verify is satisfied by the API's implementation. Using two different specifications would introduce a possible source of errors—namely, that the program relies on some property of the API that isn't satisfied by the implementation. On the other hand, using the more accurate specification of module *Lock* leads to a larger state space, making model checking less effective. This could cause us to miss an error that we would have found had we checked larger instances of the system with the simpler lock specification.

Deciding whether to use the same specification for different purposes or to write different specifications requires engineering judgment.

References

[1] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[2] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003.

[1]Remember that TLA specifications automatically allow stuttering steps, so the specification of module *SimpleLock* would not change if we changed the enabling condition of the $Aquire(t)$ action to $owner = NoThread$.