Distributed Algorithms in TLA$^+$

Leslie Lamport
Compaq Systems Research Center

How to Specify an Algorithm

How to Specify an Algorithm

Informal:

English, pseudo-code, handwaving

How to Specify an Algorithm

Informal:

English, pseudo-code, handwaving

Semi-formal:

Unity, I/O Automata, CCS, CSP, TLA

How to Specify an Algorithm

Informal:

English, pseudo-code, handwaving

Semi-formal:

Unity, I/O Automata, CCS, CSP, TLA

Temporal Logic of Actions

How to Specify an Algorithm

Informal:

English, pseudo-code, handwaving

Semi-formal:

Unity, I/O Automata, CCS, CSP, TLA

Formal language:

TLA$^+$

TLA

The basic idea:

Use *ordinary* math as much as possible.

TLA

The basic idea:

Use *ordinary* math as much as possible.

Ordinary math is simpler than any programming language.

I will describe TLA$^+$ with a simple example:

I will describe TLA$^+$ with a simple example:
the distributed mutual exclusion algorithm from

Time, Clocks, and the Ordering of Events
in a Distributed System

First, write a specification.

First, write a specification.

State the problem before giving the solution.

The module is the basic unit of a TLA$^+$ spec.

CONSTANT $Proc$

$Proc$ represents the set of processors.

CONSTANT $Proc$

CONSTANT : does not change during an execution.

CONSTANT $Proc$

CONSTANT : does not change during an execution.

A parameter of the specification.

CONSTANT $Proc$

VARIABLE $state$

VARIABLE : can change during an execution.

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$state[p]$ is the state of processor p:

"idle"	in noncritical section
"waiting"	requested resource
"owner"	has acquired resource

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

A meaningless decoration

A specification describes a set of legal behaviors.

A specification describes a set of legal behaviors.

Behavior: an infinite sequence of states

A specification describes a set of legal behaviors.

Behavior: an infinite sequence of states

State: an assignment of values to variables

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$\vdots$

$Spec \triangleq \ldots$

Specification is a single TLA formula.

$Spec \triangleq Init \wedge \Box[Next]_{state} \wedge Liveness$

Init

specifies the initial value of $state$.

$Spec \triangleq Init \land \Box[Next]_{state} \land Liveness$

$Next$

specifies how the value of $state$ changes

from one state to the next.

$$Spec \triangleq Init \land \Box[Next]_{state} \land Liveness$$

- $\Box\ Next$

  asserts that every step satisfies $Next$.

$Spec \triangleq Init \wedge \Box[Next]_{state} \wedge Liveness$

$[Next]_{state}$

equals $Next \vee$ ($state$ is unchanged).

$Spec \triangleq Init \land \Box[Next]_{state} \land Liveness$

$\Box[Next]_{state}$

asserts that every step satisfies $Next$

or leaves $state$ unchanged.

$$Spec \triangleq Init \wedge \Box[Next]_{state} \wedge Liveness$$

$Spec$ allows "stuttering steps" that leave $state$ unchanged.

$Spec \triangleq Init \wedge \Box[Next]_{state} \wedge Liveness$

$Spec$ allows "stuttering steps" that leave $state$ unchanged.

We'll see later why such steps are necessary.

$$Spec \triangleq Init \wedge \Box[Next]_{state} \wedge Liveness$$

$Liveness$ specifies what must eventually happen.

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$Init \triangleq state = \ldots$

$Init$ specifies the initial value of $state$.

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$Init \triangleq state = \ldots$

$state$ is

an array indexed by $Proc$

of {“idle”, “waiting”, “owner”}

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$Init \triangleq state = \ldots$

$state$ is

~~an array indexed by $Proc$~~

~~of {"idle", "waiting", "owner"}~~

This is programming.

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$Init \triangleq state = \ldots$

$state$ is

a function from $Proc$

to {"idle", "waiting", "owner"}

TLA is mathematics.

Functions in TLA$^+$

TLA$^+$ notation for functions.

Functions in TLA$^+$

$f[e]$

f applied to e

DOMAIN f

The domain of f.

Functions in TLA$^+$

$[S \to T]$

The set of all functions

from S (with domain S)

to T (with range a subset of T).

Functions in TLA$^+$

$[x \in S \mapsto e(x)]$

The function f with domain S such that $f[v] = e(v)$ for all v in S.

$[f$ EXCEPT $![c] = e]$

The function $\widehat{f}$ that is the same as f except with $\widehat{f}[c] = e$.

$[f$ EXCEPT $![c] = e]$

equals $[\, x \in \text{DOMAIN}\, f \mapsto \text{IF } x = c \text{ THEN } e$
$\text{ELSE } f[x]\,]$

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$Init \triangleq state = [p \in Proc \mapsto \text{“idle”}]$

Initially, $state[p] = \text{“idle”}$ for all p in $Proc$.

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$Init \triangleq state = [p \in Proc \mapsto \text{“idle”}]$

$\vdots$

$Next \triangleq \ldots$

$Next \triangleq \exists\, p \in Proc : \ldots$

. . . describes steps of process p.

$$Next \triangleq \exists\, p \in Proc : \begin{array}{l} \vee \\ \vee \\ \vee \end{array}$$

$$\begin{array}{l} \vee\ A \\ \vee\ B \\ \vee\ C \end{array} \quad \text{equals} \quad A \vee B \vee C.$$

$$Next \triangleq \exists\, p \in Proc : \lor \quad \lor \quad \lor$$

$$\begin{array}{l} \lor\ A \\ \lor\ B \\ \lor\ C \end{array} \quad \text{equals} \quad A \lor B \lor C.$$

Indentation is used to eliminate parentheses.

$$Next \triangleq \exists\, p \in Proc : \lor \lor \lor$$

$$\begin{array}{l} \land\ A \\ \land\ B \\ \land\ C \end{array} \quad \text{equals} \quad A \land B \land C.$$

Similar notation for conjunction.

$$Next \triangleq \exists\, p \in Proc : \lor\ Request(p)$$
$$\lor$$
$$\lor$$

A step satisfying $Request(p)$ is one in which p requests the resource.

$$Next \triangleq \exists\, p \in Proc : \lor\ Request(p)$$
$$\lor\ Acquire(p)$$
$$\lor$$

A step satisfying $Acquire(p)$ is one in which p acquires the resource.

$$Next \triangleq \exists\, p \in Proc : \lor\ Request(p) \lor\ Acquire(p) \lor\ Release(p)$$

A step satisfying $Release(p)$ is one in which p releases the resource.

$$Next \triangleq \exists\, p \in Proc : \lor\ Request(p) \lor\ Acquire(p) \lor\ Release(p)$$

A $Next$ step is one in which some process p requests, acquires, or releases the resource.

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$Init \triangleq state = [p \in Proc \mapsto \text{“idle”}]$

$Request(p) \triangleq \ldots$

We now define $Request(p)$.

$Request(p)$ must be true of a step

iff the step represents p requesting the resource

$Request(p)$ must be true of a step

iff the step represents p requesting the resource

iff $state[p]$ changes from "idle" to "waiting" and $state[q]$ is unchanged for all $q \neq p$

$Request(p)$ must be true of a step

iff the step represents p requesting the resource

iff $state[p]$ changes from "idle" to "waiting" and
$state[q]$ is unchanged for all $q \neq p$

iff $\land\ state_{old}[p] = \text{"idle"}$
$\land\ state_{new} = [state_{old}\ \text{EXCEPT}\ ![p] = \text{"waiting"}]$

$Request(p)$ must be true of a step

iff the step represents p requesting the resource

iff $state[p]$ changes from "idle" to "waiting" and
$state[q]$ is unchanged for all $q \neq p$

iff $\land\ state[p] = \text{"idle"}$
$\land\ state' = [state \text{ EXCEPT } ![p] = \text{"waiting"}]$

$state_{old} \leftarrow state$

$state_{new} \leftarrow state'$

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$Acquire(p) \triangleq$

$Acquire(p) \triangleq$

$\wedge\ state[p] =$ “waiting”

p is waiting.

$Acquire(p) \triangleq$

$\wedge\ state[p] = \text{“waiting”}$

$\wedge\ \forall\, q \in Proc : state[q] \neq \text{“owner”}$

No process owns the resource.

$Acquire(p) \triangleq$

$\land\ state[p] = \text{“waiting”}$

$\land\ \forall\, q \in Proc : state[q] \neq \text{“owner”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“owner”}]$

$state[p]$ is set to “owner”

$Release(p) \triangleq$

$\quad \wedge\ state[p] =$ "owner"

p is the owner.

$Release(p) \triangleq$

$\land\ state[p] = \text{“owner”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$

$state[p]$ is set to “idle”.

$$
\begin{aligned}
&Release(p) \triangleq \\
&\quad \land\ state[p] = \text{“owner”} \\
&\quad \land\ state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ \exists\, p \in Proc :\ &\lor\ Request(p) \\
&\lor\ Acquire(p) \\
&\lor\ Release(p)
\end{aligned}
$$

$$Release(p) \triangleq$$
$$\quad \land\ state[p] = \text{“owner”}$$
$$\quad \land\ state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$$

$$Next \triangleq \exists\, p \in Proc : \lor\ Request(p)$$
$$\qquad\qquad\qquad\qquad\qquad\ \lor\ Acquire(p)$$
$$\qquad\qquad\qquad\qquad\qquad\ \lor\ Release(p)$$

$$Liveness \triangleq \ldots$$

Liveness Requirements

Liveness Requirements

Deadlock-Freedom

Whenever the resource is free and some process is waiting, then some process eventually acquires it.

Liveness Requirements

Deadlock-Freedom

Whenever the resource is free and some process is waiting, then some process eventually acquires it.

Starvation-Freedom

If the resource is always released after it is acquired, then every waiting process eventually acquires it.

Liveness in TLA is usually expressed with fairness.

Liveness in TLA is usually expressed with fairness.

Two classic types of fairness: weak and strong.

Weak Fairness

Weak Fairness of Action A :

If A is ever continuously enabled,

then an A step eventually occurs.

Weak Fairness

Weak Fairness of Action A :

A cannot be enabled forever

without an A step eventually occuring.

Weak Fairness

Weak Fairness of Action A :

If A is ever enabled forever,

then an A step eventually occurs.

Weak Fairness

Weak Fairness of Action A :

If A is ever enabled forever,

then an A step eventually occurs.

After the A step, A is still enabled forever,

so another A step eventually occurs.

Weak Fairness

Weak Fairness of Action A:

If A is ever enabled forever,

then an A step eventually occurs.

After the A step, A is still enabled forever,

so another A step eventually occurs,

after which A is still enabled forever, so...

Weak Fairness

Weak Fairness of Action A:

If A is ever enabled forever,

then infinitely many A steps occur.

Weak Fairness

Weak Fairness of Action A:

If A is eventually enabled forever,

then infinitely many A steps occur.

Weak Fairness

Weak Fairness of Action A :

$$\Diamond\Box(\text{ENABLED}\ A) \Rightarrow \Box\Diamond A$$

Weak Fairness

Weak Fairness of Action A :

$$\Diamond\Box(\text{ENABLED } A) \Rightarrow \Box\Diamond A$$

ENABLED A $\triangleq$ there exist values of the primed variables that satisfy A

Weak Fairness

Weak Fairness of Action A :

$$\Box\Diamond(\neg \text{ENABLED}\, A) \;\vee\; \Box\Diamond A$$

A is disabled infinitely often

or infinitely many A steps occur.

Weak Fairness

$$\text{WF}\ (A)\ \triangleq\ \Diamond\Box(\text{ENABLED}\ \ A\ \ )\ \Rightarrow\ \Box\Diamond\ A$$

Weak Fairness in TLA

$$\mathsf{WF}_v(A) \triangleq \Diamond\Box(\mathsf{ENABLED}\,\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v$$

where $\langle A\rangle_v \triangleq A \wedge (v' \neq v)$

Weak Fairness in TLA

$$\mathrm{WF}_v(A) \triangleq \Diamond\Box(\mathrm{ENABLED}\,\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v$$

where $\langle A\rangle_v \triangleq A \wedge (v' \neq v)$

v is usually the tuple of all variables, so

$v' \neq v$ iff the value of some variable changes.

Weak Fairness in TLA

$$\text{WF}_v(A) \triangleq \Diamond\Box(\text{ENABLED}\,\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v$$

where $\langle A\rangle_v \triangleq A \wedge (v' \neq v)$

v is usually the tuple of all variables, so an $\langle A\rangle_v$ step is a nonstuttering A step.

Weak Fairness in TLA

$$\mathrm{WF}_v(A) \triangleq \Diamond\Box(\mathrm{ENABLED}\,\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v$$

where $\langle A\rangle_v \triangleq A \wedge (v' \neq v)$

$\langle\ \rangle_v$ needed so stuttering steps make no difference.

Strong Fairness

Strong Fairness of Action A:

If A is ever continually enabled

(perhaps continually disabled too),

then an A step eventually occurs.

Strong Fairness

Strong Fairness of Action A:

If A is infinitely often enabled,

then infinitely many A steps occur.

Strong Fairness

Strong Fairness of Action A :

$\Box\Diamond(\text{ENABLED}\ A) \Rightarrow \Box\Diamond A$

Strong Fairness

Strong Fairness of Action A :

$$\Box\Diamond(\text{ENABLED } A) \;\Rightarrow\; \Box\Diamond A$$

Weak Fairness of Action A :

$$\Diamond\Box(\text{ENABLED } A) \;\Rightarrow\; \Box\Diamond A$$

Strong Fairness

Strong Fairness of Action A :

$$\Box\Diamond(\text{ENABLED } A) \Rightarrow \Box\Diamond A$$

Weak Fairness of Action A :

$$\Diamond\Box(\text{ENABLED } A) \Rightarrow \Box\Diamond A$$

Strong fairness is stronger than (implies) weak fairness.

Deadlock-Freedom

Whenever the resource is free and some process is waiting, then some process eventually acquires it.

Deadlock-Freedom

Whenever the resource is free and some process is waiting, then some process eventually acquires it.

Some process acquires resource:

$\exists\, p \in Proc \;:\; Acquire(p)$

Deadlock-Freedom

Whenever the resource is free and some process is waiting, then some process eventually acquires it.

Some process acquires resource:

$\exists\, p \in Proc \;:\; Acquire(p)$

Resource is free and some process is waiting:

ENABLED $(\exists\, p \in Proc \;:\; Acquire(p))$

Deadlock-Freedom

Whenever the resource is free and some process is waiting, then some process eventually acquires it.

Weak fairness of $\exists\, p \in Proc : Acquire(p)$

Starvation-Freedom

If the resource is always released after it is acquired, then every waiting process eventually acquires it.

Starvation-Freedom

If the resource is always released after it is acquired, then no process can wait forever.

Starvation-Freedom

For each p, if the resource is always released after it is acquired, then p cannot wait forever.

Starvation-Freedom

For each p, if the resource is continually released while p is waiting, then p must eventually acquire it.

Starvation-Freedom

For each p, if $Acquire(p)$ is continually enabled, then an $Acquire(p)$ step must eventually occur.

Starvation-Freedom

For each p, strong fairness of $Acquire(p)$.

$$StarvationFree \triangleq \forall\, p \in Proc : \mathsf{SF}_{state}(Acquire(p))$$

$StarvationFree \triangleq \forall\, p \in Proc : \mathsf{SF}_{state}(Acquire(p))$

$Spec \triangleq Init \wedge \Box[Next]_{state} \wedge StarvationFree$

We take starvation freedom as our liveness condition.

$$StarvationFree \triangleq \forall\, p \in Proc : \mathsf{SF}_{state}(Acquire(p))$$

$$Spec \triangleq Init \wedge \Box[Next]_{state} \wedge StarvationFree$$

This ends module $Mutex$.

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$$Init \triangleq state = [p \in Proc \mapsto \text{“idle”}]$$

$$Request(p) \triangleq \land state[p] = \text{“idle”} \\ \land state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$$

$$Acquire(p) \triangleq \land state[p] = \text{“waiting”} \\ \land \forall q \in Proc : state[q] \neq \text{“owner”} \\ \land state' = [state \text{ EXCEPT } ![p] = \text{“owner”}]$$

$$Release(p) \triangleq \land state[p] = \text{“owner”} \\ \land state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$$

$$Next \triangleq \exists p \in Proc : Request(p) \lor Acquire(p) \lor Release(p)$$

$$StarvationFree \triangleq \forall p \in Proc : \mathsf{SF}_{state}(Acquire(p))$$

$$Spec \triangleq Init \land \Box[Next]_{state} \land StarvationFree$$

MODULE *Mutex*

CONSTANT *Proc*

VARIABLE *state*

$$Init \triangleq state = [p \in Proc \mapsto \text{“idle”}]$$

$$Request(p) \triangleq \land\ state[p] = \text{“idle”} \\ \land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$$

$$Acquire(p) \triangleq \land\ state[p] = \text{“waiting”} \\ \land\ \forall\, q \in Proc : state[q] \neq \text{“owner”} \\ \land\ state' = [state \text{ EXCEPT } ![p] = \text{“owner”}]$$

$$Release(p) \triangleq \land\ state[p] = \text{“owner”} \\ \land\ state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$$

$$Next \triangleq \exists\, p \in Proc : Request(p) \lor Acquire(p) \lor Release(p)$$

$$StarvationFree \triangleq \forall\, p \in Proc : \mathsf{SF}_{state}(Acquire(p))$$

$$Spec \triangleq Init \land \Box[Next]_{state} \land StarvationFree$$

Formalism Overhead

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$$Init \triangleq state = [p \in Proc \mapsto \text{“idle”}]$$

$$Request(p) \triangleq \land state[p] = \text{“idle”} \\ \land state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$$

$$Acquire(p) \triangleq \land state[p] = \text{“waiting”} \\ \land \forall q \in Proc : state[q] \neq \text{“owner”} \\ \land state' = [state \text{ EXCEPT } ![p] = \text{“owner”}]$$

$$Release(p) \triangleq \land state[p] = \text{“owner”} \\ \land state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$$

$$Next \triangleq \exists p \in Proc : Request(p) \lor Acquire(p) \lor Release(p)$$

$$StarvationFree \triangleq \forall p \in Proc : \mathsf{SF}_{state}(Acquire(p))$$

$$Spec \triangleq Init \land \Box[Next]_{state} \land StarvationFree$$

Temporal Logic

MODULE *Mutex*

CONSTANT $Proc$

VARIABLE $state$

$$Init \triangleq state = [p \in Proc \mapsto \text{“idle”}]$$

$$Request(p) \triangleq \land state[p] = \text{“idle”} \\ \land state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$$

$$Acquire(p) \triangleq \land state[p] = \text{“waiting”} \\ \land \forall q \in Proc : state[q] \neq \text{“owner”} \\ \land state' = [state \text{ EXCEPT } ![p] = \text{“owner”}]$$

$$Release(p) \triangleq \land state[p] = \text{“owner”} \\ \land state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$$

$$Next \triangleq \exists p \in Proc : Request(p) \lor Acquire(p) \lor Release(p)$$

$$StarvationFree \triangleq \forall p \in Proc : \mathsf{SF}_{state}(Acquire(p))$$

$$Spec \triangleq Init \land \Box[Next]_{state} \land StarvationFree$$

Ordinary Math

The complexity is where it belongs:

in the ordinary math.

The complexity is where it belongs:

in the ordinary math.

Mutual exclusion spec: 11 lines

Ordinary math: 80%

(ignoring overhead)

The complexity is where it belongs:

in the ordinary math.

Mutual exclusion spec: 11 lines

Ordinary math: 80%

Algorithm spec: 70 lines

Ordinary math: 95%

The complexity is where it belongs:

in the ordinary math.

Mutual exclusion spec: 11 lines

Ordinary math: 80%

Algorithm spec: 70 lines

Ordinary math: 95%

System spec: 2000 lines

Ordinary math: $>$ 99%

Checking Properties of the Specification

The best way to find errors in a specification is to check that it satisfies desired properties.

EXTENDS $Mutex$

Imports declarations and definitions from $Mutex$.

MODULE *MutexProperties*

EXTENDS $Mutex$

$MutualExclusion \triangleq$

$\forall\, p, q \in Proc :$

$(p \neq q) \Rightarrow (\{state[p], state[q]\} \neq \{\text{“owner”}\})$

True iff no two processes both own the resource.

MODULE *MutexProperties*

EXTENDS $Mutex$

$$MutualExclusion \triangleq$$
$$\forall\, p, q \in Proc :$$
$$(p \neq q) \Rightarrow (\{state[p], state[q]\} \neq \{\text{“owner”}\})$$

THEOREM $Spec \Rightarrow \Box MutualExclusion$

Asserts that the specification implies mutual exclusion.

MODULE *MutexProperties*

EXTENDS $Mutex$

$$MutualExclusion \triangleq$$
$$\forall\, p, q \in Proc :$$
$$(p \neq q) \Rightarrow (\{state[p], state[q]\} \neq \{\text{“owner”}\})$$

THEOREM $Spec \Rightarrow \Box MutualExclusion$

In TLA, *satisfies* is $\Rightarrow$.

$$EventuallyAcquires \triangleq$$
$$\forall\, p \in Proc : (state[p] = \text{“waiting”}) \rightsquigarrow (state[p] = \text{“owner”})$$

$P \rightsquigarrow Q \quad \triangleq \quad \Box(P \Rightarrow \Diamond Q)$

$\equiv$ whenever P is true, Q is true then or later.

$$EventuallyAcquires \triangleq$$
$$\forall\, p \in Proc : (state[p] = \text{“waiting”}) \rightsquigarrow (state[p] = \text{“owner”})$$

Every waiting process eventually acquires the resource.

$$EventuallyAcquires \triangleq$$
$$\forall\, p \in Proc : (state[p] = \text{“waiting”}) \rightsquigarrow (state[p] = \text{“owner”})$$

THEOREM $Spec \Rightarrow EventuallyAcquires$

Proving Properties in TLA

Proof means math.

Proving Properties in TLA

Proof means math.

In TLA, there's no translation from the spec to math.

Proving Properties in TLA

Proof means math.

In TLA, there's no translation from the spec to math.

For example, proving invariance:

$$Inv \wedge Next \Rightarrow Inv'$$

Proving Properties in TLA

Proof means math.

In TLA, there's no translation from the spec to math.

For example, proving invariance:

$$Inv \wedge Next \Rightarrow Inv'$$

TLA is as good as it gets for formalizing assertional reasoning.

You can write as rigorous (or as nonrigorus)
a TLA proof as you want.

You can write as rigorous (or as nonrigorus)
a TLA proof as you want.

I will discuss tools, not proofs today.

You can write as rigorous (or as nonrigorus)
a TLA proof as you want.

I will discuss tools, not proofs today.

Tools are no substitute for proofs,

You can write as rigorous (or as nonrigorus)
a TLA proof as you want.

I will discuss tools, not proofs today.

Tools are no substitute for proofs,
but they're more fun.

The TLC Model Checker

(Can also run simulations.)

Model checking and simulation can catch simple mistakes.

The TLC Model Checker

Model checking and simulation can catch simple mistakes.

Model checking usually can't handle a large enough model to find subtle errors,

The TLC Model Checker

Model checking and simulation can catch simple mistakes.

Model checking usually can't handle a large enough model to find subtle errors, and simulation is notoriously unreliable.

The TLC Model Checker

Model checking and simulation can catch simple mistakes.

Model checking usually can't handle a large enough model to find subtle errors, and simulation is notoriously unreliable.

Use TLC first, prove later.

The TLC Model Checker

Model checking and simulation can catch simple mistakes.

Model checking usually can't handle a large enough model to find subtle errors, and simulation is notoriously unreliable.

Can use TLC to check proof.

TLA$^+$ in ASCII

Tools and human writers use ASCII.

TLA$^+$ in ASCII

Tools and human writers use ASCII.

See handout for ASCII $\leftrightarrow$ math translation.

```
------------------------ MODULE Mutex ------------------------
CONSTANT Proc
VARIABLE state
------------------------------------------------------------
Init == state = [p \in Proc |-> "idle"]

Request(p) == /\ state[p] = "idle"
              /\ state' = [state EXCEPT ![p] = "waiting"]

Acquire(p) == /\ state[p] = "waiting"
              /\ \A q \in Proc : state[q] # "owner"
              /\ state' = [state EXCEPT ![p] = "owner"]

Release(p) == /\ state[p] = "owner"
              /\ state' = [state EXCEPT ![p] = "idle"]

Next == \E p \in Proc : Request(p) \/ Acquire(p) \/ Release(p)
------------------------------------------------------------
StarvationFree == \A p \in Proc : SF_state(Acquire(p))

Spec == Init /\ [][Next]_state /\ StarvationFree
============================================================
```

Input to TLC

- The specification

Input to TLC

- The specification
- A configuration file defining a finite model.

The Configuration File

Instantiates constant parameters.

Example: $Proc$

The Configuration File

Instantiates constant parameters.

Can specify an arbitrary constraint.

Example: $\forall\, r \in Link\ :\ Len(msgQueue[r]) < 4$

The Configuration File

Instantiates constant parameters.

Can specify an arbitrary constraint.

Names formulas $Init$, $Next$, $Liveness$.

The Configuration File

Instantiates constant parameters.

Can specify an arbitrary constraint.

Names formulas $Init$, $Next$, $Liveness$.

Specifies properties to check.

The Configuration File for *MutexProperties*

The Configuration File for *MutexProperties*

```
CONSTANTS Proc = {p1, p2, p3}
```

The Configuration File for *MutexProperties*

```
CONSTANTS Proc = {p1, p2, p3}

INIT Init

NEXT Next

TEMPORAL StarvationFree
```

The Configuration File for *MutexProperties*

```
CONSTANTS Proc = {p1, p2, p3}
INIT Init
NEXT Next
TEMPORAL StarvationFree
INVARIANT MutualExclusion
IMPLIED-TEMPORAL EventuallyAcquires
```

TLC Demo

$$EventuallyAcquires \triangleq$$
$$\forall\, p \in Proc : (state[p] = \text{“waiting”}) \rightsquigarrow (state[p] = \text{“owner”})$$

~~THEOREM $Spec \Rightarrow EventuallyAcquires$~~

$Spec$ does not require that a process ever release the resource.

$$EventuallyAcquires \triangleq$$
$$\forall\, p \in Proc : (state[p] = \text{“waiting”}) \rightsquigarrow (state[p] = \text{“owner”})$$

$$AlwaysReleases \triangleq \forall\, p \in Proc : \mathsf{WF}_{state}(Release(p))$$

THEOREM $Spec \wedge AlwaysReleases \Rightarrow EventuallyAcquires$

The Algorithm

Let's go directly to TLA$^+$.

MODULE *TimeClocks*

EXTENDS $Naturals$

Imports $Naturals$ module, which defines Nat, $+$, ...

MODULE *TimeClocks*

EXTENDS *Naturals*, *Sequences*

Imports *Sequences* module, which defines:

Append(*s*, *e*)

MODULE *TimeClocks*

EXTENDS *Naturals*, *Sequences*

Imports *Sequences* module, which defines:

Append(s, e), *Head*(s), *Tail*(s), ...

MODULE *TimeClocks*

EXTENDS *Naturals*, *Sequences*

Defines sequences to be tuples, so $\langle\rangle$ is the empty sequence.

MODULE *TimeClocks*

EXTENDS *Naturals*, *Sequences*

CONSTANTS *Proc*,

Same as in *Mutex* spec.

MODULE *TimeClocks*

EXTENDS *Naturals*, *Sequences*

CONSTANTS *Proc*, $\_ \ll \_$

$\ll$ is an arbitrary ordering of processes.

MODULE *TimeClocks*

EXTENDS $Naturals,\ Sequences$

CONSTANTS $Proc,\ \_ \ll \_$

ASSUME $\forall\, p \in Proc\ :$

$$\begin{aligned}
&\land\ \neg(p \ll p)\\
&\land\ \forall\, q \in Proc \setminus \{p\}\ :\ (p \ll q)\ \lor\ (q \ll p)\\
&\land\ \forall\, q,\ r \in Proc\ :\ (p \ll q)\ \land\ (q \ll r) \Rightarrow (p \ll r)
\end{aligned}$$

Asserts that $\ll$ totally orders $Proc$.

$a \prec b \triangleq$

Request a has smaller timestamp than request b,
or has the same timestamp and $\ll$ processor.

$a \prec b \triangleq$

A request is a record with:

- TS field that is a number (a timestamp)
- $proc$ field that is a process

$$a \prec b \triangleq \lor\ a.TS < b.TS$$
$$\lor\ (a.TS = b.TS) \land (a.proc \ll b.proc)$$

A request is a record with:

- TS field that is a number (a timestamp)
- $proc$ field that is a process

VARIABLES $state$,

As in $Mutex$ spec:

$state[p] \in \{$ “idle” , “waiting” , “owner” $\}$

VARIABLES $state$, $msgQ$,

$msgQ[p][q]$ is the queue of messages in transit from p to q.

VARIABLES $state,\ msgQ,\ reqSet,$

$reqSet[p]$ is the set of current requests known by p.

VARIABLES $state,\ msgQ,\ reqSet,\ clock,$

$clock[p]$ is p's (logical) clock.

VARIABLES $state$, $msgQ$, $reqSet$, $clock$, $lastTSent$,

$lastTSent[p][q]$ is the timestamp of the last message sent by p to q.

VARIABLES $state$, $msgQ$, $reqSet$, $clock$, $lastTSent$, $lastTRcvd$

$lastTRcvd[p][q]$ is the timestamp of the last message received by p from q.

VARIABLES $state,\ msgQ,\ reqSet,\ clock,\ lastTSent,\ lastTRcvd$

$vars \triangleq \langle state, msgQ, reqSet, clock, lastTSent, lastTRcvd \rangle$

The tuple of all declared variables.

$$
\begin{aligned}
Init \triangleq\ & \\
& \land\ state = \\
& \land\ msgQ = \\
& \land\ reqSet = \\
& \land\ clock = \\
& \land\ lastTSent = \\
& \land\ lastTRcvd =
\end{aligned}
$$

$$
\begin{array}{lll}
Init & \triangleq & \\
\quad \wedge\ state & = & [p \in Proc \mapsto \text{“idle”}] \\
\quad \wedge\ msgQ & = & \\
\quad \wedge\ reqSet & = & \\
\quad \wedge\ clock & = & \\
\quad \wedge\ lastTSent & = & \\
\quad \wedge\ lastTRcvd & = &
\end{array}
$$

$Init \triangleq$

$\land\ state = [p \in Proc \mapsto$ "idle"]

$\land\ msgQ = [p \in Proc \mapsto \quad]$

$\land\ reqSet =$

$\land\ clock =$

$\land\ lastTSent =$

$\land\ lastTRcvd =$

$msgQ[p][q] = \langle\,\rangle$, for $q \neq p$

$$
\begin{array}{lll}
Init & \triangleq & \\
\quad \wedge\ state & = & [p \in Proc \mapsto \text{“idle”}] \\
\quad \wedge\ msgQ & = & [p \in Proc \mapsto [q \in Proc \setminus \{p\} \mapsto \langle\,\rangle]] \\
\quad \wedge\ reqSet & = & \\
\quad \wedge\ clock & = & \\
\quad \wedge\ lastTSent & = & \\
\quad \wedge\ lastTRcvd & = &
\end{array}
$$

$msgQ[p][q] = \langle\,\rangle$, for $q \neq p$

$$
\begin{array}{lll}
Init & \triangleq & \\
\quad \wedge\ state & = & [p \in Proc \mapsto \text{“idle”}] \\
\quad \wedge\ msgQ & = & [p \in Proc \mapsto [q \in Proc \setminus \{p\} \mapsto \langle\rangle]] \\
\quad \wedge\ reqSet & = & [p \in Proc \mapsto \{\}] \\
\quad \wedge\ clock & = & \\
\quad \wedge\ lastTSent & = & \\
\quad \wedge\ lastTRcvd & = &
\end{array}
$$

$$
\begin{array}{lll}
Init & \triangleq & \\
\quad \wedge\ state & = & [p \in Proc \mapsto \text{“idle”}] \\
\quad \wedge\ msgQ & = & [p \in Proc \mapsto [q \in Proc \setminus \{p\} \mapsto \langle\rangle]] \\
\quad \wedge\ reqSet & = & [p \in Proc \mapsto \{\}] \\
\quad \wedge\ clock & \in & [Proc \to Nat] \\
\quad \wedge\ lastTSent & = & \\
\quad \wedge\ lastTRcvd & = &
\end{array}
$$

$clock[p] \in Nat$

$$
\begin{array}{lll}
Init \triangleq & & \\
\quad \land\ state & = & [p \in Proc \mapsto \text{“idle”}] \\
\quad \land\ msgQ & = & [p \in Proc \mapsto [q \in Proc \setminus \{p\} \mapsto \langle\rangle]] \\
\quad \land\ reqSet & = & [p \in Proc \mapsto \{\}] \\
\quad \land\ clock & \in & [Proc \to Nat] \\
\quad \land\ lastTSent & = & [p \in Proc \mapsto [q \in Proc \setminus \{p\} \mapsto 0]] \\
\quad \land\ lastTRcvd & = &
\end{array}
$$

$$
\begin{array}{lll}
Init \triangleq & & \\
\quad \land\ state & = & [p \in Proc \mapsto \text{“idle”}] \\
\quad \land\ msgQ & = & [p \in Proc \mapsto [q \in Proc \setminus \{p\} \mapsto \langle\rangle]] \\
\quad \land\ reqSet & = & [p \in Proc \mapsto \{\}] \\
\quad \land\ clock & \in & [Proc \to Nat] \\
\quad \land\ lastTSent & = & [p \in Proc \mapsto [q \in Proc \setminus \{p\} \mapsto 0]] \\
\quad \land\ lastTRcvd & = & [p \in Proc \mapsto [q \in Proc \setminus \{p\} \mapsto 0]]
\end{array}
$$

$Request(p) \triangleq$

Process p requests the resource.

$Request(p) \triangleq$

$\quad \wedge \ state[p] =$ "idle"

As before, $state[p]$ must equal "idle" . . .

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

. . . and $state[p]$ is set to “waiting”.

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$\land\ \exists\, n \in Nat :$

$\quad \land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$clock[p]$ is set to some new value n.

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$\land\ \exists\, n \in Nat\ :$

$\land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$\land\ n > clock[p]$

The new value of $clock[p]$ must be $>$ its old value.

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$\land\ \exists\, n \in Nat\ :$

$\land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$\land\ n > clock[p]$

$\land$ LET $msg \triangleq$

IN

Defines msg locally in the IN clause.

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$\land\ \exists\, n \in Nat\ :$

$\land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$\land\ n > clock[p]$

$\land$ LET $msg \triangleq [TS \mapsto n,\ proc \mapsto p,\ cmd \mapsto \text{“acquire”}]$

IN

msg is a record with three components:

$msg.TS = n$

$msg.proc = p$

$msg.cmd =$ “acquire”

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$\land\ \exists\, n \in Nat\ :$

$\land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$\land\ n > clock[p]$

$\land$ LET $msg \triangleq [TS \mapsto n,\ proc \mapsto p,\ cmd \mapsto \text{“acquire”}]$

IN $\land\ msgQ' =$

Append msg to $msgQ[p][q]$, for each q (other than p).

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$\land\ \exists\, n \in Nat\ :$

$\land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$\land\ n > clock[p]$

$\land$ LET $msg \triangleq [TS \mapsto n,\ proc \mapsto p,\ cmd \mapsto \text{“acquire”}]$

IN $\land\ msgQ' = [msgQ \text{ EXCEPT } ![p] =$

$[q \in Proc \setminus \{p\} \mapsto Append(msgQ[p][q],$

$msg)]]$

Append msg to $msgQ[p][q]$, for each q (other than p).

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$\land\ \exists\, n \in Nat\ :$

$\land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$\land\ n > clock[p]$

$\land$ LET $msg \triangleq [TS \mapsto n,\ proc \mapsto p,\ cmd \mapsto \text{“acquire”}]$

IN $\land\ msgQ' = [msgQ \text{ EXCEPT } ![p] =$

$[q \in Proc \setminus \{p\} \mapsto Append(@[q],\ msg)]\,]$

@ denotes $msgQ[p]$.

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$\land\ \exists\, n \in Nat :$

$\land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$\land\ n > clock[p]$

$\land$ LET $msg \triangleq [TS \mapsto n,\ proc \mapsto p,\ cmd \mapsto \text{“acquire”}]$

IN $\land\ msgQ' = [msgQ \text{ EXCEPT } ![p] =$

$[q \in Proc \setminus \{p\} \mapsto Append(@[q],\ msg)]\,]$

$\land\ reqSet' = [reqSet \text{ EXCEPT } ![p] = @ \cup \{msg\}]$

Put msg in the set $reqSet[p]$.

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$\land\ \exists\, n \in Nat :$

$\land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$\land\ n > clock[p]$

$\land$ LET $msg \triangleq [TS \mapsto n,\ proc \mapsto p,\ cmd \mapsto \text{“acquire”}]$

IN $\land\ msgQ' = [msgQ \text{ EXCEPT } ![p] =$
$[q \in Proc \setminus \{p\} \mapsto Append(@[q],\ msg)]]$

$\land\ reqSet' = [reqSet \text{ EXCEPT } ![p] = @ \cup \{msg\}]$

$\land\ lastTSent' = [lastTSent \text{ EXCEPT } ![p] = [q \in Proc \setminus \{p\} \mapsto n]]$

Update $lastTSent[p][q]$.

$Request(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$\land\ \exists\, n \in Nat :$

$\land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$\land\ n > clock[p]$

$\land$ LET $msg \triangleq [TS \mapsto n,\ proc \mapsto p,\ cmd \mapsto \text{“acquire”}]$

IN $\land\ msgQ' = [msgQ \text{ EXCEPT } ![p] =$

$[q \in Proc \setminus \{p\} \mapsto Append(@[q],\ msg)]]$

$\land\ reqSet' = [reqSet \text{ EXCEPT } ![p] = @ \cup \{msg\}]$

$\land\ lastTSent' = [lastTSent \text{ EXCEPT } ![p] = [q \in Proc \setminus \{p\} \mapsto n]]$

$\land$ UNCHANGED $lastTRcvd$

$Acquire(p) \triangleq$

Process p acquires the resource.

$Acquire(p) \triangleq$

LET $pReq \triangleq$

IN

$pReq$ is the request by p in $reqSet[p]$.

$Acquire(p) \triangleq$

LET $pReq \triangleq$ CHOOSE $req \in reqSet[p] : req.proc = p$

IN

CHOOSE known to mathematicians as Hilbert's ε.

$Acquire(p) \triangleq$

LET $pReq \triangleq$ CHOOSE $req \in reqSet[p] : req.proc = p$

IN $\wedge\ state[p] =$ "waiting"

p must be waiting.

$Acquire(p) \triangleq$

LET $pReq \triangleq$ CHOOSE $req \in reqSet[p] : req.proc = p$

IN $\land\ state[p] =$ "waiting"

$\land\ \forall\, req \in reqSet[p] \setminus \{pReq\} : pReq \prec req$

$pReq$ must be the earliest request in $reqSet[p]$.

$Acquire(p) \triangleq$

LET $pReq \triangleq$ CHOOSE $req \in reqSet[p] : req.proc = p$

IN $\land\ state[p] =$ "waiting"

$\land\ \forall\, req \in reqSet[p] \setminus \{pReq\} : pReq \prec req$

$\land\ \forall\, q \in Proc \setminus \{p\} :$

p must have received a message with a later timestamp than $pReq$'s from every other processor.

$Acquire(p) \triangleq$

LET $pReq \triangleq$ CHOOSE $req \in reqSet[p] : req.proc = p$

IN $\land\ state[p] =$ "waiting"

$\land\ \forall\, req \in reqSet[p] \setminus \{pReq\} : pReq \prec req$

$\land\ \forall\, q \in Proc \setminus \{p\} :$

$pReq \prec [TS \mapsto lastTRcvd[p][q],\ proc \mapsto q]$

A record with two components:

TS component equals $lastTRcvd[p][q]$

$proc$ component equals q

$Acquire(p) \triangleq$

LET $pReq \triangleq$ CHOOSE $req \in reqSet[p] : req.proc = p$

IN $\land\ state[p] =$ "waiting"

$\land\ \forall\, req \in reqSet[p] \setminus \{pReq\} : pReq \prec req$

$\land\ \forall\, q \in Proc \setminus \{p\} :$

$pReq \prec [TS \mapsto lastTRcvd[p][q]+1,\ proc \mapsto q]$

The next request p receives from every other processor must have a later timestamp than $pReq$'s.

$Acquire(p) \triangleq$

LET $pReq \triangleq$ CHOOSE $req \in reqSet[p] : req.proc = p$

IN $\land\ state[p] =$ "waiting"

$\land\ \forall\, req \in reqSet[p] \setminus \{pReq\} : pReq \prec req$

$\land\ \forall\, q \in Proc \setminus \{p\} :$

$pReq \prec [TS \mapsto lastTRcvd[p][q] + 1,\ proc \mapsto q]$

$\land\ state' = [state$ EXCEPT $![p] =$ "owner"$]$

$Acquire(p) \triangleq$

LET $pReq \triangleq$ CHOOSE $req \in reqSet[p] : req.proc = p$

IN $\land\ state[p] =$ "waiting"

$\land\ \forall\, req \in reqSet[p] \setminus \{pReq\} : pReq \prec req$

$\land\ \forall\, q \in Proc \setminus \{p\} :$

$pReq \prec [TS \mapsto lastTRcvd[p][q] + 1,\ proc \mapsto q]$

$\land\ state' = [state$ EXCEPT $![p] =$ "owner"$]$

$\land\ reqSet' = [reqSet$ EXCEPT $![p] = @ \setminus \{pReq\}]$

Remove p's request from $reqSet[p]$.

$Acquire(p) \triangleq$

LET $pReq \triangleq$ CHOOSE $req \in reqSet[p] : req.proc = p$

IN $\land\ state[p] =$ "waiting"

$\land\ \forall\, req \in reqSet[p] \setminus \{pReq\} : pReq \prec req$

$\land\ \forall\, q \in Proc \setminus \{p\} :$

$pReq \prec [TS \mapsto lastTRcvd[p][q] + 1,\ proc \mapsto q]$

$\land\ state' = [state$ EXCEPT $![p] =$ "owner"$]$

$\land\ reqSet' = [reqSet$ EXCEPT $![p] = @ \setminus \{pReq\}]$

Remove p's request from $reqSet[p]$.

Could just as well be done by $Release(p)$.

$$Acquire(p) \triangleq$$

LET $pReq \triangleq$ CHOOSE $req \in reqSet[p] : req.proc = p$

IN $\land\ state[p] =$ "waiting"

$\land\ \forall\, req \in reqSet[p] \setminus \{pReq\} : pReq \prec req$

$\land\ \forall\, q \in Proc \setminus \{p\} :$

$$pReq \prec [TS \mapsto lastTRcvd[p][q] + 1,\ proc \mapsto q]$$

$\land\ state' = [state$ EXCEPT $![p] =$ "owner"$]$

$\land\ reqSet' = [reqSet$ EXCEPT $![p] = @ \setminus \{pReq\}]$

$\land$ UNCHANGED $\langle msgQ,\ clock,\ lastTSent,\ lastTRcvd \rangle$

$Release(p) \triangleq$

p releases the resource.

$Release(p) \triangleq$

$\land \ state[p] =$ "owner"

$Release(p) \triangleq$

$\land\ state[p] = \text{“owner”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$

$Release(p) \triangleq$

$\land\ state[p] = \text{“owner”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$

$\land$ LET $msg \triangleq [TS \mapsto clock[p],\ proc \mapsto p,\ cmd \mapsto \text{“release”}]$

IN

$Release(p) \triangleq$

$\land\ state[p] = \text{“owner”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$

$\land$ LET $msg \triangleq [TS \mapsto clock[p],\ proc \mapsto p,\ cmd \mapsto \text{“release”}]$

IN $msgQ' = [msgQ \text{ EXCEPT } ![p] =$

$[q \in Proc \setminus \{p\} \mapsto Append(@[q],\ msg)]]$

Append msg to $msgQ[p][q]$ for all $p \neq q$.

$Release(p) \triangleq$

$\land\ state[p] = \text{“owner”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$

$\land$ LET $msg \triangleq [TS \mapsto clock[p],\ proc \mapsto p,\ cmd \mapsto \text{“release”}]$

IN $msgQ' = [msgQ \text{ EXCEPT } ![p] =$

$[q \in Proc \setminus \{p\} \mapsto Append(@[q],\ msg)]]$

$\land\ lastTSent' = [lastTSent \text{ EXCEPT } ![p] =$

$[q \in Proc \setminus \{p\} \mapsto clock[p]]]$

Update $lastTSent[p][q]$ for all $q \neq p$.

$Release(p) \triangleq$

$\land\ state[p] = \text{“owner”}$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$

$\land$ LET $msg \triangleq [TS \mapsto clock[p],\ proc \mapsto p,\ cmd \mapsto \text{“release”}]$

IN $msgQ' = [msgQ \text{ EXCEPT } ![p] =$

$[q \in Proc \setminus \{p\} \mapsto Append(@[q],\ msg)]]$

$\land\ lastTSent' = [lastTSent \text{ EXCEPT } ![p] =$

$[q \in Proc \setminus \{p\} \mapsto clock[p]]]$

$\land$ UNCHANGED $\langle clock,\ lastTRcvd,\ reqSet \rangle$

$RcvMsg(p,\, q) \triangleq$

p receives a message from q.

$RcvMsg(p,\, q) \triangleq$

LET $msg \triangleq Head(msgQ[q][p])$

The next message from q to p.

$RcvMsg(p,\ q) \triangleq$

LET $msg \triangleq Head(msgQ[q][p])$

$msgQTail \triangleq [msgQ \text{ EXCEPT } ![q][p] = Tail(@)]$

$msgQ$ with the head of $msgQ[q][p]$ removed.

$RcvMsg(p,\ q) \triangleq$

LET $msg \triangleq Head(msgQ[q][p])$

$msgQTail \triangleq [msgQ \text{ EXCEPT } ![q][p] = Tail(@)]$

$ack \triangleq [TS \mapsto clock'[p],\ proc \mapsto p,\ cmd \mapsto$ "ack"]

An ack message that might be sent.

IN $\land\ msgQ[q][p] \neq \langle\,\rangle$

The message queue is nonempty.

IN $\land\ msgQ[q][p] \neq \langle\,\rangle$

$\land\ clock' = [clock \text{ EXCEPT } ![p] =$

IF $msg.TS > @$ THEN $msg.TS$ ELSE $@\,]$

$clock[p]$ is set to the maximum of its old value
and $msg.TS$.

IN $\land\ msgQ[q][p] \neq \langle\,\rangle$

$\land\ clock' = [clock \text{ EXCEPT } ![p] =$

IF $msg.TS > @$ THEN $msg.TS$ ELSE $@\,]$

$\land$ IF

THEN

ELSE

When must p send an ack message to q?

IN $\land$ $msgQ[q][p] \neq \langle\,\rangle$

$\land$ $clock' = [clock$ EXCEPT $![p] =$

IF $msg.TS > @$ THEN $msg.TS$ ELSE $@\,]$

$\land$ IF $\land$ $msg.cmd =$ "acquire"

THEN

ELSE

If q sent an "acquire" message. . .

IN $\land$ $msgQ[q][p] \neq \langle\,\rangle$

$\land$ $clock' = [clock$ EXCEPT $![p] =$

IF $msg.TS > @$ THEN $msg.TS$ ELSE $@\,]$

$\land$ IF $\land$ $msg.cmd =$ "acquire"

$\land$ $[TS \mapsto lastTSent[p][q] + 1,\ proc \mapsto p] \prec msg$

THEN

ELSE

. . . and p has not sent a message to q with a large enough timestamp to enable $Acquire(q)$.

IN $\land\ msgQ[q][p] \neq \langle\,\rangle$

$\land\ clock' = [clock$ EXCEPT $![p] =$

IF $msg.TS > @$ THEN $msg.TS$ ELSE $@\,]$

$\land$ IF $\land\ msg.cmd =$ "acquire"

$\land\ [TS \mapsto lastTSent[p][q] + 1,\ proc \mapsto p] \prec msg$

THEN $\land\ msgQ' = [msgQTail$ EXCEPT

$![p][q] = Append(@,\ ack)]$

ELSE

Then append ack to $msgQ[p][q]$ . . .

IN $\land\ msgQ[q][p] \neq \langle\,\rangle$

$\land\ clock' = [clock$ EXCEPT $![p] =$

IF $msg.TS > @$ THEN $msg.TS$ ELSE $@\,]$

$\land$ IF $\land\ msg.cmd =$ "acquire"

$\land\ [TS \mapsto lastTSent[p][q] + 1,\ proc \mapsto p] \prec msg$

THEN $\land\ msgQ' = [msgQTail$ EXCEPT

$![p][q] = Append(@,\ ack)]$

$\land\ lastTSent' = [lastTSent$ EXCEPT

$![p][q] = clock'[p]\,]$

ELSE

. . . and update $lastTSent[p][q]$.

IN $\land\ msgQ[q][p] \neq \langle\,\rangle$

$\land\ clock' = [clock$ EXCEPT $![p] =$

IF $msg.TS > @$ THEN $msg.TS$ ELSE $@\,]$

$\land$ IF $\land\ msg.cmd =$ "acquire"

$\land\ [TS \mapsto lastTSent[p][q] + 1,\ proc \mapsto p] \prec msg$

THEN $\land\ msgQ' = [msgQTail$ EXCEPT

$![p][q] = Append(@, ack)]$

$\land\ lastTSent' = [lastTSent$ EXCEPT

$![p][q] = clock'[p]\,]$

ELSE $\land\ msgQ' = msgQTail$

Else just remove head of $msgQ[q][p]$.

IN $\land\ msgQ[q][p] \neq \langle\,\rangle$

$\land\ clock' = [clock$ EXCEPT $![p] =$

IF $msg.TS > @$ THEN $msg.TS$ ELSE $@\,]$

$\land$ IF $\land\ msg.cmd =$ "acquire"

$\land\ [TS \mapsto lastTSent[p][q] + 1,\ proc \mapsto p] \prec msg$

THEN $\land\ msgQ' = [msgQTail$ EXCEPT

$![p][q] = Append(@,\ ack)]$

$\land\ lastTSent' = [lastTSent$ EXCEPT

$![p][q] = clock'[p]\,]$

ELSE $\land\ msgQ' = msgQTail$

$\land$ UNCHANGED $lastTSent$

$\land\ lastTRcvd' = [lastTRcvd \text{ EXCEPT } ![p][q] = msg.TS]$

Update $lastTRcvd[p][q]$.

$\land\ lastTRcvd' = [lastTRcvd \text{ EXCEPT } ![p][q] = msg.TS]$

$\land\ reqSet' = [reqSet \text{ EXCEPT } ![p] =$

CASE $msg.cmd =$ "acquire" $\rightarrow$

$\square\ msg.cmd =$ "release" $\rightarrow$

$\square\ msg.cmd =$ "ack" $\rightarrow$

New value of $reqSet[p]$ depends on message type.

$\land\ lastTRcvd' = [lastTRcvd \text{ EXCEPT } ![p][q] = msg.TS]$

$\land\ reqSet' = [reqSet \text{ EXCEPT } ![p] =$

CASE $msg.cmd =$ "acquire" $\rightarrow$ @ $\cup\ \{msg\}$

$\square\ msg.cmd =$ "release" $\rightarrow$

$\square\ msg.cmd =$ "ack" $\rightarrow$

Add an "acquire" to $reqSet[p]$.

$\land\ lastTRcvd' = [lastTRcvd \text{ EXCEPT } ![p][q] = msg.TS]$

$\land\ reqSet' = [reqSet \text{ EXCEPT } ![p] =$

CASE $msg.cmd =$ "acquire" $\rightarrow$ @ $\cup\ \{msg\}$

$\square$ $msg.cmd =$ "release" $\rightarrow$

$\{m \in$ @ $:\ m.proc \neq q\}$

$\square$ $msg.cmd =$ "ack" $\rightarrow$

For a "release" message, remove q's request from $reqSet[p]$.

$\land\ lastTRcvd' = [lastTRcvd \text{ EXCEPT } ![p][q] = msg.TS]$

$\land\ reqSet' = [reqSet \text{ EXCEPT } ![p] =$

CASE $msg.cmd =$ "acquire" $\rightarrow$ @ $\cup\ \{msg\}$

$\square\ msg.cmd =$ "release" $\rightarrow$

$\{m \in$ @ $:\ m.proc \neq q\}$

$\square\ msg.cmd =$ "ack" $\rightarrow$ @]

An "ack" messages leaves $reqSet[p]$ unchanged.

$\land\ lastTRcvd' = [lastTRcvd \text{ EXCEPT } ![p][q] = msg.TS]$

$\land\ reqSet' = [reqSet \text{ EXCEPT } ![p] =$

$\text{CASE } msg.cmd = \text{“acquire”} \rightarrow @ \cup \{msg\}$

$\quad \square\ msg.cmd = \text{“release”} \rightarrow$

$\qquad \{m \in @ : m.proc \neq q\}$

$\quad \square\ msg.cmd = \text{“ack”} \rightarrow @\ ]$

$\land$ UNCHANGED $state$

$Tick(p) \triangleq$

Advance p's clock.

$$
\begin{array}{l}
Tick(p) \triangleq \\
\quad \land\ \exists\, n \in Nat\ :\ \land\ n > clock[p] \\
\qquad\qquad\qquad\ \land\ clock' = [clock\ \texttt{EXCEPT}\ ![p] = n]
\end{array}
$$

$Tick(p) \triangleq$

$\land\ \exists\, n \in Nat : \land\ n > clock[p]$

$\land\ clock' = [clock \text{ EXCEPT } ![p] = n]$

$\land$ UNCHANGED $\langle state, msgQ, reqSet, lastTSent, lastTRcvd \rangle$

$$
\begin{array}{l}
Next \triangleq \\
\quad \exists\, p \in Proc : \lor\ Request(p) \lor Acquire(p) \lor Release(p) \\
\qquad\qquad\quad \lor\ \exists\, q \in Proc \setminus \{p\} : RcvMsg(p,\, q) \\
\qquad\qquad\quad \lor\ Tick(p)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Liveness} \triangleq \\
\quad \forall\, p \in \mathit{Proc} \,:\, \wedge\, \mathsf{WF}_{\mathit{vars}}(\mathit{Acquire}(p)) \\
\qquad\qquad\quad \wedge\, \forall\, q \in \mathit{Proc} \setminus \{p\} \,:\, \mathsf{WF}_{\mathit{vars}}(\mathit{RcvMsg}(p,\, q))
\end{array}
$$

$$Liveness \triangleq$$
$$\forall\, p \in Proc : \wedge\, \mathsf{WF}_{vars}(Acquire(p))$$
$$\wedge\, \forall\, q \in Proc \setminus \{p\} : \mathsf{WF}_{vars}(RcvMsg(p, q))$$

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge Liveness$$

$$
\begin{array}{l}
\mathit{Liveness} \triangleq \\
\quad \forall\, p \in \mathit{Proc} \,:\, \wedge\, \mathsf{WF}_{\mathit{vars}}(\mathit{Acquire}(p)) \\
\qquad\qquad\quad \wedge\, \forall\, q \in \mathit{Proc} \setminus \{p\} \,:\, \mathsf{WF}_{\mathit{vars}}(\mathit{RcvMsg}(p,\, q)) \\
\\
\mathit{Spec} \triangleq \mathit{Init} \wedge \Box[\mathit{Next}]_{\mathit{vars}} \wedge \mathit{Liveness}
\end{array}
$$

TLA$^+$ Versus the Original

TLA$^+$ Versus the Original

The 1978 Paper

TLA$^+$ Versus the Original

The 1978 Paper

- Explains the idea

TLA$^+$ Versus the Original

The 1978 Paper

- Explains the idea
- Sketches an algorithm

TLA$^+$ Versus the Original

The 1978 Paper

- Explains the idea
- Sketches an algorithm

The TLA$^+$ Specification

TLA$^+$ Versus the Original

The 1978 Paper

- Explains the idea
- Sketches an algorithm

The TLA$^+$ Specification

- Specifies a real algorithm

TLA$^+$ Versus the Original

The 1978 Paper

- Explains the idea
- Sketches an algorithm

The TLA$^+$ Specification

- Specifies a real algorithm
- Fills in the missing details

TLA$^+$ Versus the Original

The 1978 Paper

- Explains the idea
- Sketches an algorithm

The TLA$^+$ Specification

- Specifies a real algorithm
- Fills in the missing details
- Not a substitute for explaining the idea

MODULE *TimeClocksCorrectness*

The $TimeClocks$ algorithm implements the $Mutex$ spec.

MODULE *TimeClocksCorrectness*

EXTENDS $TimeClocks$

MODULE *TimeClocksCorrectness*

EXTENDS $TimeClocks,\ Mutex$

MODULE *TimeClocksCorrectness*

EXTENDS $TimeClocks$, ~~$Mutex$~~

Name conflicts: $Init$, $Request$, ...

MODULE *TimeClocksCorrectness*

EXTENDS $TimeClocks$

$M \triangleq$ INSTANCE $Mutex$

Imports definitions from $Mutex$ with renaming.

MODULE *TimeClocksCorrectness*

EXTENDS *TimeClocks*

$M \triangleq$ INSTANCE *Mutex*

$Init \rightarrow M\,!\,Init, \quad Request \rightarrow M\,!\,Request, \quad \ldots$

MODULE *TimeClocksCorrectness*

EXTENDS $TimeClocks$

$M \triangleq$ INSTANCE $Mutex$

THEOREM $Spec \Rightarrow M\,!\,Spec$

Every behavior satisfying $Spec$ (of $TimeClocks$) satisfies $M\,!\,Spec$ ($Spec$ of $Mutex$).

MODULE *TimeClocksCorrectness*

EXTENDS $TimeClocks$

$M \triangleq$ INSTANCE $Mutex$

THEOREM $Spec \Rightarrow M!Spec$

$RcvMsg(p, q)$ and $Tick(p)$ steps are stuttering steps of $Mutex$ spec.

Checking Correctness with TLC

Checking Correctness with TLC

Problem: There are infinitely many reachable states.

Checking Correctness with TLC

Problem: There are infinitely many reachable states.

$clock[p]$ can equal any natural number.

Checking Correctness with TLC

Problem: There are infinitely many reachable states.
$clock[p]$ can equal any natural number.

Solution: Substitute a finite set of numbers
for Nat.

MODULE *MC_TimeClocksCorrectness*

EXTENDS *TimeClocksCorrectness*

MODULE *MC_TimeClocksCorrectness*

EXTENDS *TimeClocksCorrectness*

CONSTANT *NumOfNats*

$MC\_Nat \triangleq 0 \,..\, NumOfNats$

Replace *Nat* with $\{0, \ldots, NumOfNats\}$

MODULE *MC_TimeClocksCorrectness*

EXTENDS *TimeClocksCorrectness*

CONSTANT *NumOfNats*, *NumOfProcs*

$MC\_Nat \triangleq 0 \,..\, NumOfNats$

$MC\_Proc \triangleq 1 \,..\, NumOfProcs$

Instantiate $\prec$ with $<$.

MODULE *MC_TimeClocksCorrectness*

EXTENDS *TimeClocksCorrectness*

CONSTANT *NumOfNats*, *NumOfProcs*

$MC\_Nat \triangleq 0 \,..\, NumOfNats$

$MC\_Proc \triangleq 1 \,..\, NumOfProcs$

$MC\_MNext \triangleq [M\,!\,Next]_{state}$

TLC will check $Init \wedge \Box Next \Rightarrow \Box[M\,!\,Next]_{state}$

The Configuration File

The Configuration File

```
CONSTANTS NumOfNats  = 5
          NumOfProcs = 2
```

The Configuration File

```
CONSTANTS NumOfNats  = 5
          NumOfProcs = 2
          Nat  <- MC_Nat
          Proc <- MC_Proc
```

<- means replace with defined symbol.

The Configuration File

```
CONSTANTS NumOfNats  = 5
          NumOfProcs = 2
          Nat  <- MC_Nat
          Proc <- MC_Proc
          \ll  <- <
```

\ll is ASCII for $\ll$.

The Configuration File

```
CONSTANTS NumOfNats  = 5
          NumOfProcs = 2
          Nat  <- MC_Nat
          Proc <- MC_Proc
          \ll  <- <
INIT Init
NEXT Next
TEMPORAL Liveness
```

The Configuration File

```
CONSTANTS NumOfNats  = 5
          NumOfProcs = 2
          Nat  <- MC_Nat
          Proc <- MC_Proc
          \ll  <- <
INIT Init
NEXT Next
TEMPORAL Liveness
IMPLIED-ACTION MC_MNext
```

Check $Init \wedge \Box Next \Rightarrow \Box MC\_MNext$

The Configuration File

```
CONSTANTS NumOfNats  = 5
          NumOfProcs = 2
          Nat  <- MC_Nat
          Proc <- MC_Proc
          \ll  <- <
INIT Init
NEXT Next
TEMPORAL Liveness
IMPLIED-ACTION MC_MNext
IMPLIED-TEMPORAL M!StarvationFree
```

Check $Init \wedge \Box Next \wedge Liveness \Rightarrow M\,!\,StarvationFree$

Running TLC produces:

```
...
Error: deadlock reached. The behavior...
STATE 1:
/\ clock = (1 :> 5 @@ 2 :> 5)
/\ state = (1 :> "idle" @@ 2 :> "idle")
/\ lastTRcvd = (1 :> (2 :> 0) @@ 2 :> (1 :> 0))
/\ msgQ = (1 :> (2 :> <<  >>) @@ 2 :> (1 :> <<  >>))
/\ reqSet = (1 :> {} @@ 2 :> {})
/\ lastTSent = (1 :> (2 :> 0) @@ 2 :> (1 :> 0))
```

```
STATE 1:
/\ clock = (1 :> 5 @@ 2 :> 5)
```

$clock[p]$ can't advance past $NumOfNats$

Can disable deadlock detection.

Can disable deadlock detection.

But liveness checking still won't work.

Can disable deadlock detection.

But liveness checking still won't work.

Solution: use a constraint to stop state exploration before deadlock occurs.

The Configuration File

```
CONSTANTS NumOfNats  = 5
          NumOfProcs = 2
          Nat  <- MC_Nat
          Proc <- MC_Proc
          \ll  <- <
INIT Init
NEXT Next
TEMPORAL Liveness
IMPLIED-ACTION MC_MNext
IMPLIED-TEMPORAL M!StarvationFree
CONSTRAINT Constraint
```

MODULE *MC_TimeClocksCorrectness*

EXTENDS *TimeClocksCorrectness*

CONSTANT *NumOfNats*, *NumOfProcs*

$MC\_Nat \triangleq 0 \,..\, NumOfNats$

$MC\_Proc \triangleq 1 \,..\, NumOfProcs$

$MC\_MNext \triangleq [M\,!\,Next]_{state}$

$Constraint \triangleq \forall\, p \in Proc : clock[p] < NumberOfNats$

Running TLC on a 500MHz Alpha produces

```
Model checking completed...
35863 states generated, 13322 distinct states found.
```

in 58 seconds.

Running TLC on a 650MHz Alpha produces

```
Model checking completed...
35863 states generated, 13322 distinct states found.
```

in 36 seconds.

(12 seconds without liveness checking)

Further Statistics

Number of states $\approx O(NumOfNats^{NumOfProcs^2})$

Further Statistics

Number of states $\approx O(NumOfNats^{NumOfProcs^2})$

Procs	*Nats*	*states*	*time*
2	5	13K	0:33
2	6	39K	1:53
2	7	100K	5:24
2	8	235K	14:53
2	9	512K	42:26
3	2	5K	0:18
3	3	151K	10:53
3	4	2262K	5:39:02

Model Checking

Can handle only small models.

Model Checking

Can handle only small models.

Not a substitute for proof.

Model Checking

Can handle only small models.

Not a substitute for proof.

But really easy.

Truth in Advertising

Limitations of TLC Version 1:

Truth in Advertising

Limitations of TLC Version 1:

- It doesn't implement INSTANCE .

  Must substitute by hand (it's easy).

Truth in Advertising

Limitations of TLC Version 1:

- It doesn't implement INSTANCE .
  Must substitute by hand (it's easy).

- The cfg file can't handle infix operators.
  I used $Prec(a, b)$ instead of $a \ll b$.

The State-Machine Approach

Proposed in *Time, Clocks. . .* as a general method for implementing distributed systems.

The State-Machine Approach

Proposed in *Time, Clocks. . .* as a general method for implementing distributed systems.

A distributed algorithm lets each process execute an arbitrary state machine on the same sequence of inputs.

The State-Machine Approach

Proposed in *Time, Clocks. . .* as a general method for implementing distributed systems.

A distributed algorithm lets each process execute an arbitrary state machine on the same sequence of inputs.

You implement a particular system by defining the appropriate state machine.

The State-Machine Approach

Proposed in *Time, Clocks. . .* as a general method for implementing distributed systems.

A distributed algorithm lets each process execute an arbitrary state machine on the same sequence of inputs.

Mutual exclusion was just the simplest example I could think of.

A state machine is a mapping:

$$State \times Command \rightarrow State' \times Output$$

Convenient to put parameters in a separate module.

CONSTANTS

$Proc$,

The set of processes.

CONSTANTS

$Proc$, Cmd,

The set of state machine commands

CONSTANTS

Proc, *Cmd*, *InitState*,

The initial state of the state machine.

MODULE *SMParameters*

CONSTANTS

Proc, *Cmd*, *InitState*, $NewState(\_, \_)$,

$NewState(s, c)$ is the state obtained by executing command c in state s.

MODULE *SMParameters*

CONSTANTS

Proc, *Cmd*, *InitState*, *NewState*(_, _), *Output*(_, _)

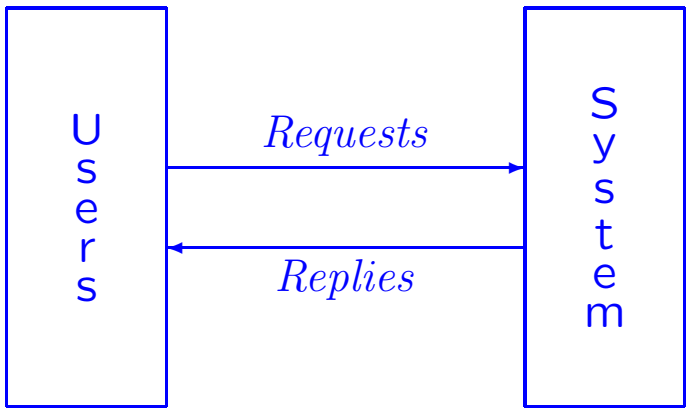
Output(s, c) is the output produced by executing command c in state s.

The Interface

We must specify the interface between the users/environment and the system.

The Interface

We must specify the interface between the users/environment and the system.



The Interface

We must specify the interface between the users/environment and the system.

We want a simple, abstract interface, with uninterpreted $Request$ and $Reply$ actions.

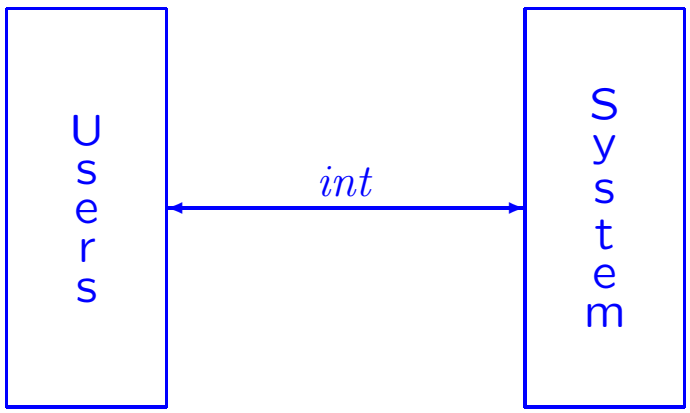
The Interface

We must specify the interface between the users/environment and the system.

We want a simple, abstract interface, with uninterpreted $Request$ and $Reply$ actions.

How do we do that with a state-based formalism like TLA?

VARIABLE int



VARIABLE int

CONSTANTS

$Request(\_, \_, \_, \_)$,

A $Request(int, int', c, p)$ step represents process p issuing command c to the system.

VARIABLE int

CONSTANTS

$Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$

A $Reply(int, int', r, p)$ step represents the system issuing reply r to process p.

VARIABLE int

CONSTANTS

$Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$

A particular interface is specified by instantiating $Request$ and $Reply$

MODULE *SMParameters*

CONSTANTS

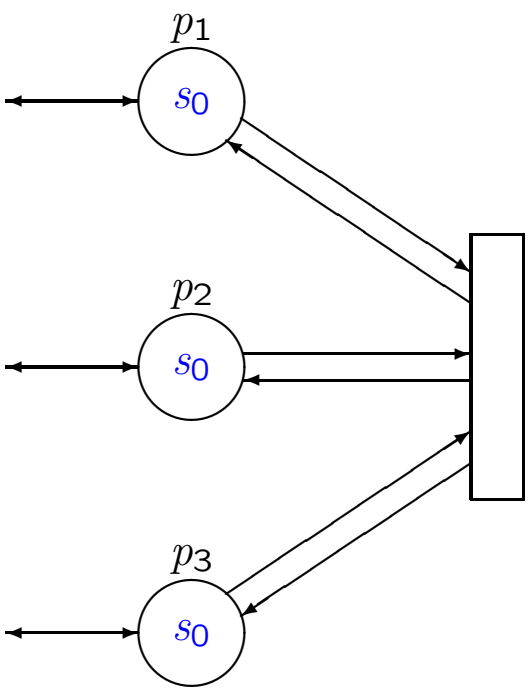
Proc, *Cmd*, *InitState*, *NewState*(_, _), *Output*(_, _)

VARIABLE *int*

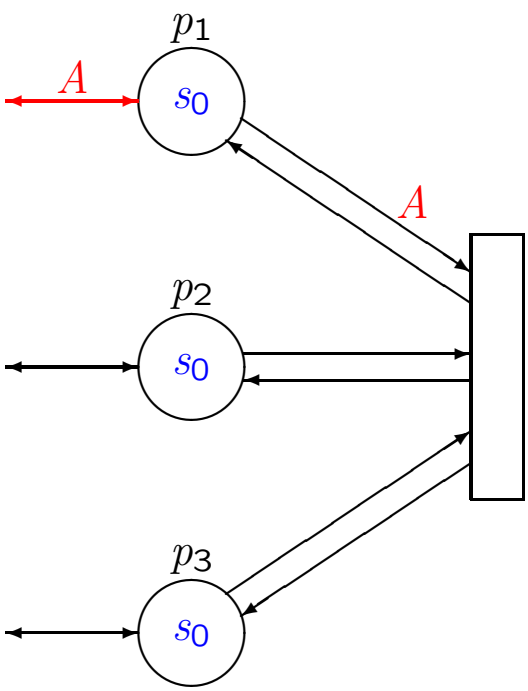
CONSTANTS

Request(_, _, _, _), *Reply*(_, _, _, _)

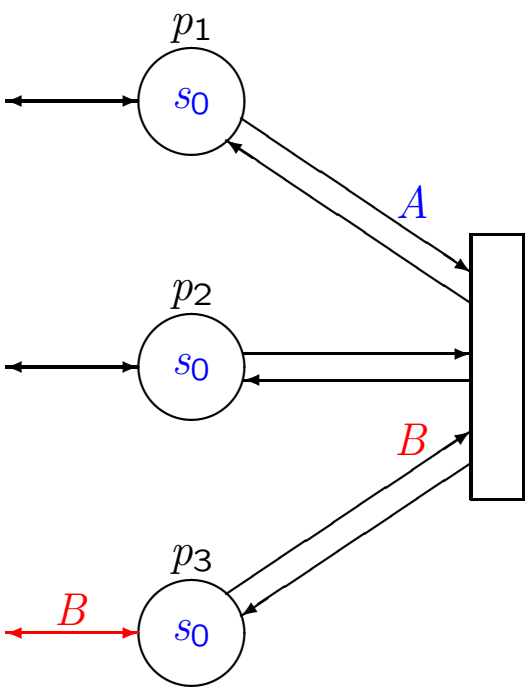
$$s_0 = InitState$$



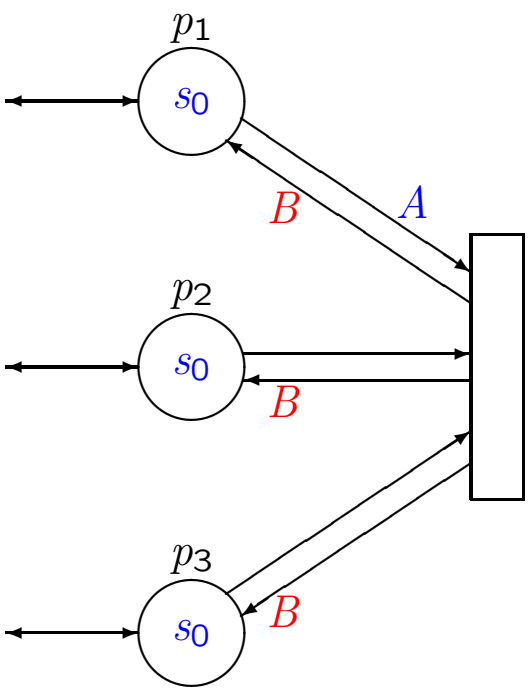
$Request(int,\ int',\ A,\ p_1)$



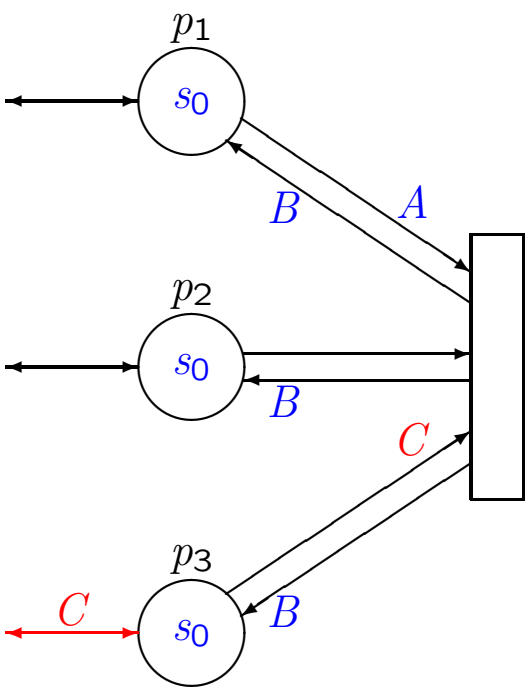
$Request(int,\ int',\ B,\ p_3)$



$SequenceCmd(p_3)$



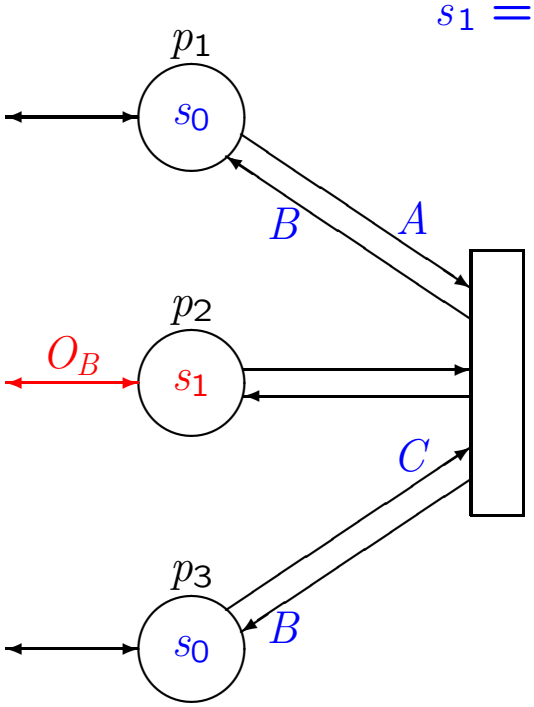
$Request(int,\ int',\ C,\ p_3)$



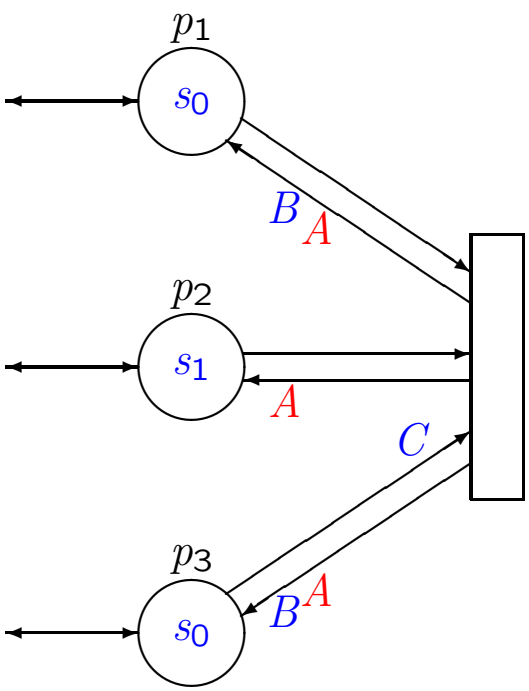
$$Reply(int,\ int',\ O_B,\ p_2)$$

$$O_B = Output(s_0,\ B)$$

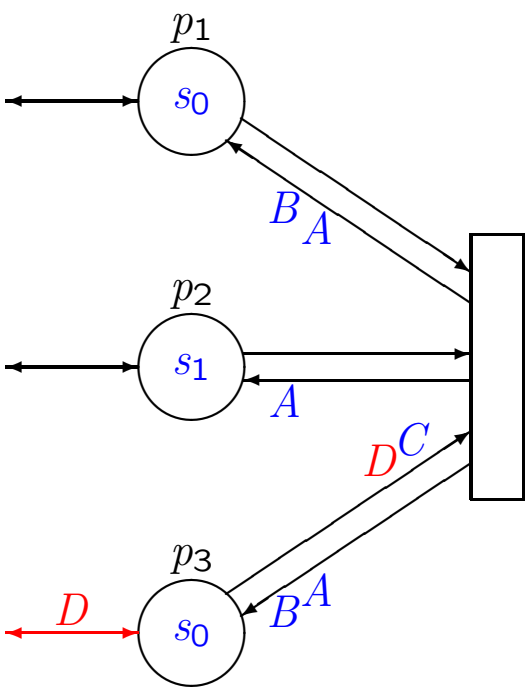
$$s_1 = NewState(s_0,\ B)$$



$SequenceCmd(p_1)$



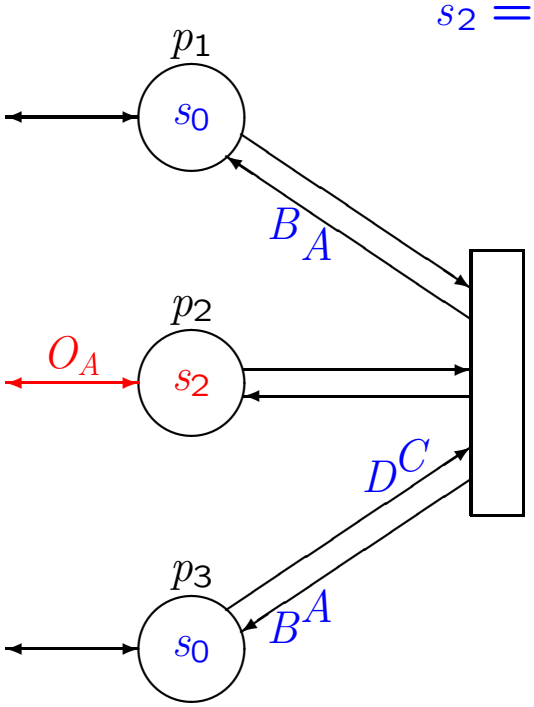
$Request(int,\ int',\ D,\ p_3)$



$$Reply(int,\ int',\ O_A,\ p_2)$$

$$O_A = Output(s_1,\ A)$$

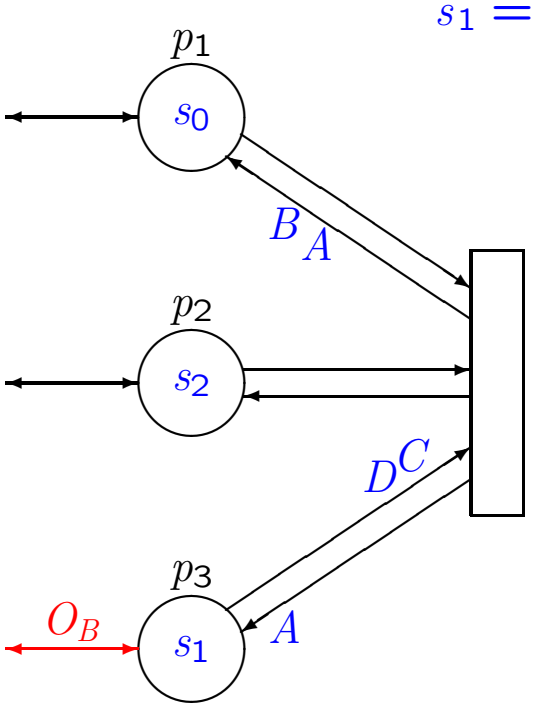
$$s_2 = NewState(s_1,\ A)$$



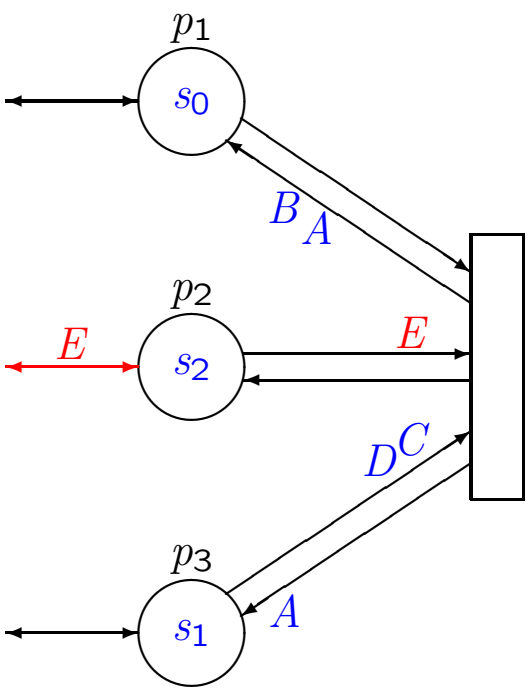
$$Reply(int,\ int',\ O_B,\ p_3)$$

$$O_B = Output(s_0,\ B)$$

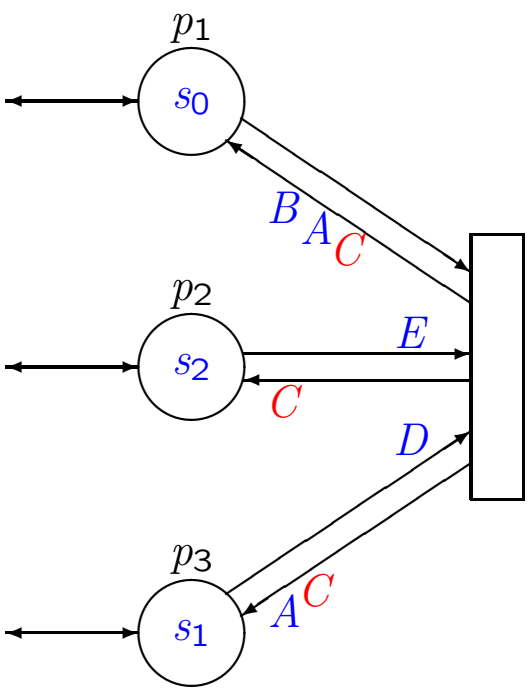
$$s_1 = NewState(s_0,\ B)$$



$Request(int,\ int',\ E,\ p_2)$



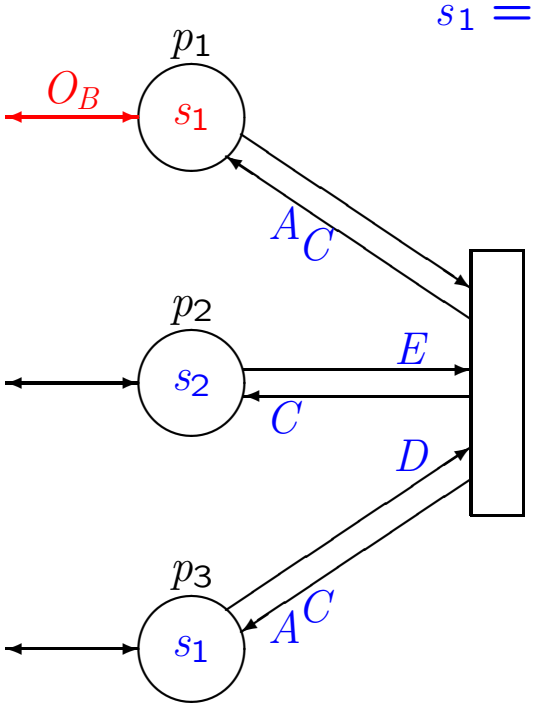
$SequenceCmd(p_3)$



$$Reply(int,\ int',\ O_B,\ p_1)$$

$$O_B = Output(s_0,\ B)$$

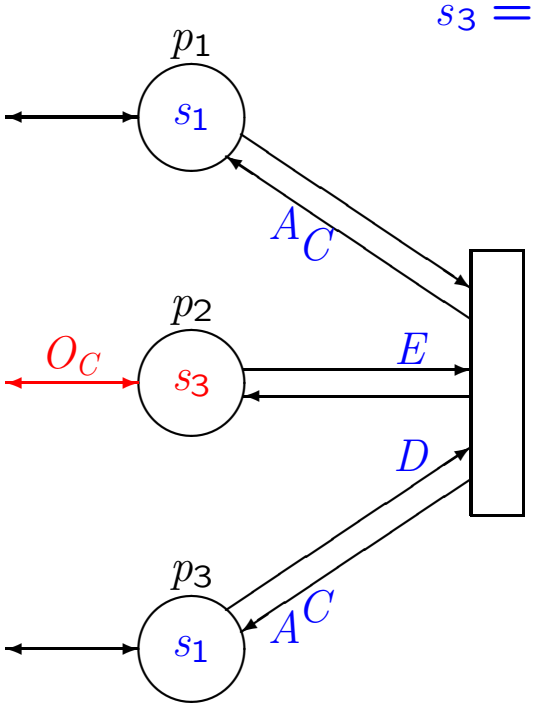
$$s_1 = NewState(s_0,\ B)$$



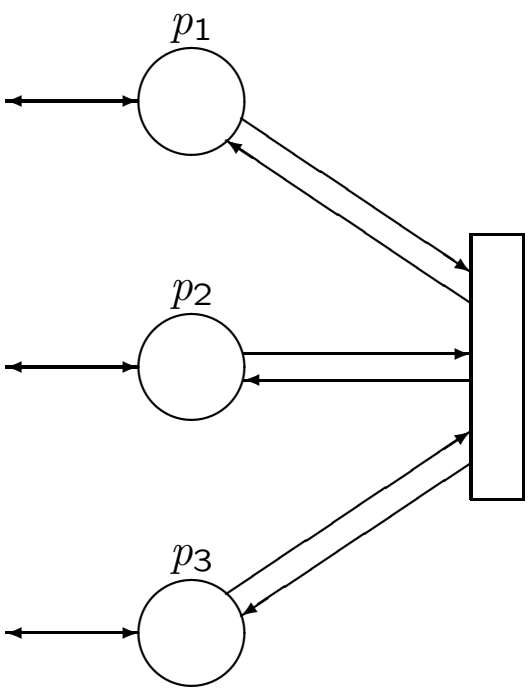
$$Reply(int,\ int',\ O_C,\ p_2)$$

$$O_C = Output(s_2,\ C)$$

$$s_3 = NewState(s_2,\ C)$$

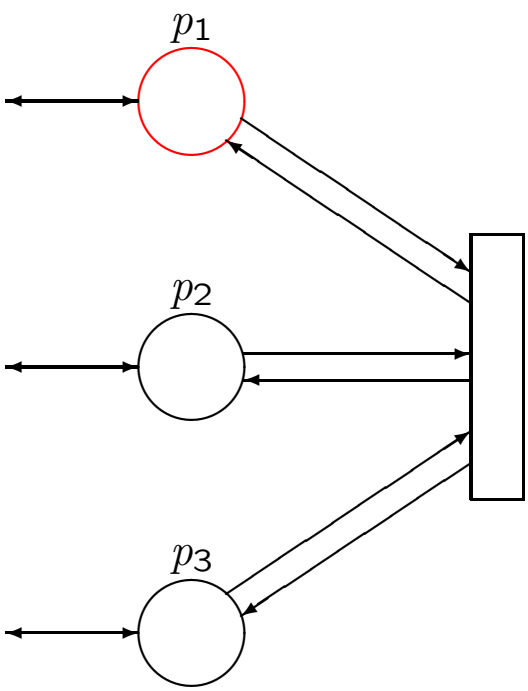


The variables:



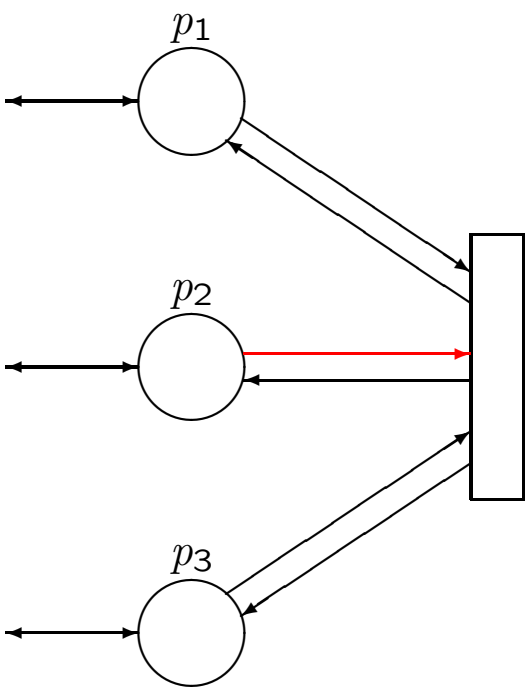
The variables:

$state[p_1]$



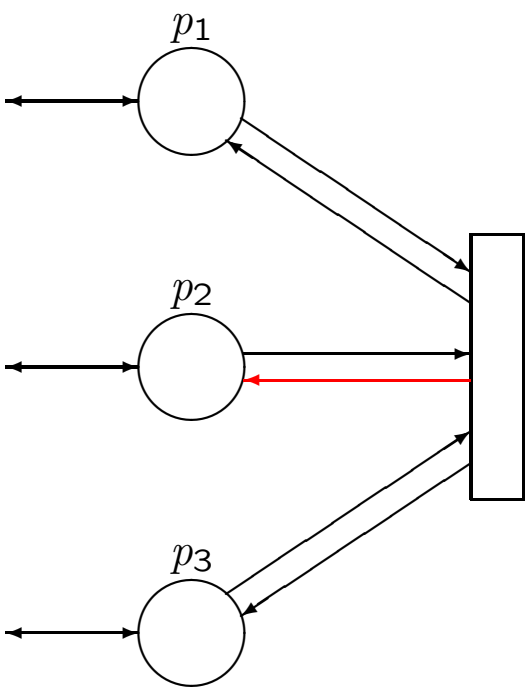
The variables:

$rcvd[p_2]$



The variables:

$pending[p_2]$



Problem:

We are specifying the interface.

Problem:

We are specifying the interface.

The interface is described by the values of int.

Problem:

We are specifying the interface.

The interface is described by the values of int.

The spec should not mention $state$, $rcvd$, or $pending$.

The solution:

Hide $state$, $rcvd$, and $pending$.

The solution:

Hide $state$, $rcvd$, and $pending$.

In mathematics, hiding is $\exists$.

The solution:

Hide $state$, $rcvd$, and $pending$.

In mathematics, hiding is $\exists$.

In temporal logic, hiding is $\exists$.

MODULE *StateMachine*

EXTENDS $SMParams$, ...

MODULE *StateMachine*

EXTENDS $SMParams, \ldots$

VARIABLES $state, pending, rcvd$

MODULE *StateMachine*

EXTENDS $SMParams, \ldots$

VARIABLES $state, pending, rcvd$

$$\vdots$$

$ISpec \triangleq \ldots$

MODULE *StateMachine*

EXTENDS $SMParams, \ldots$

VARIABLES $state, pending, rcvd$

$\vdots$

$ISpec \triangleq \ldots$

$Spec \triangleq \exists\, state, pending, rcvd : ISpec$

VARIABLES $state$, $pending$, $rcvd$

$$\vdots$$

$ISpec \triangleq \ldots\ state\ \ldots$

$Spec \triangleq \exists\, state, pending, rcvd : ISpec$

Two different declarations of $state$.

VARIABLES $state$, $pending$, $rcvd$

$\vdots$

$ISpec \triangleq \ldots\ state\ \ldots$

~~$Spec \triangleq \exists\ state, pending, rcvd : ISpec$~~

To keep you out of trouble, TLA$^+$ does not allow the re-declaration of identifiers.

MODULE *InnerSM*

EXTENDS $SMParams, \ldots$

VARIABLES $state, pending, rcvd$

$$\vdots$$

$ISpec \triangleq \ldots$

EXTENDS $SMParams$

MODULE *StateMachine*

EXTENDS *SMParams*

$Inner(state, pending, rcvd) \triangleq$ INSTANCE $InnerSM$

If $F \triangleq exp$ appears in $InnerSM$, then this defines $Inner(s, p, r)!F$ to equal

exp with $state \leftarrow s$, $pending \leftarrow p$, $rcvd \leftarrow r$.

MODULE *StateMachine*

EXTENDS *SMParams*

$Inner(state, pending, rcvd) \triangleq$ INSTANCE *InnerSM*

$Spec \triangleq \exists\, s, p, r : Inner(s, p, r)!ISpec$

MODULE *StateMachine*

EXTENDS *SMParams*

$Inner(state, pending, rcvd) \triangleq$ INSTANCE *InnerSM*

$Spec \triangleq \exists\, state, pending, rcvd :$
$Inner(state, pending, rcvd)\,!\,ISpec$

MODULE *InnerSM*

EXTENDS $SMParams$, $Sequences$

MODULE *InnerSM*

EXTENDS *SMParams*, *Sequences*

VARIABLES *state*, *pending*, *rcvd*

MODULE *InnerSM*

EXTENDS *SMParams*, *Sequences*

VARIABLES *state*, *pending*, *rcvd*

$vars \triangleq \langle int, state, pending, rcvd \rangle$

MODULE *InnerSM*

EXTENDS *SMParams*, *Sequences*

VARIABLES *state*, *pending*, *rcvd*

$vars \triangleq \langle int, state, pending, rcvd \rangle$

$Init \triangleq \land\ state = [p \in Proc \mapsto InitState]$

MODULE *InnerSM*

EXTENDS *SMParams*, *Sequences*

VARIABLES *state*, *pending*, *rcvd*

$vars \triangleq \langle int, state, pending, rcvd \rangle$

$$Init \triangleq \begin{array}{ll} \land & state = [p \in Proc \mapsto InitState] \\ \land & pending = [p \in Proc \mapsto \langle\,\rangle] \end{array}$$

MODULE *InnerSM*

EXTENDS *SMParams*, *Sequences*

VARIABLES *state*, *pending*, *rcvd*

$$vars \triangleq \langle int, state, pending, rcvd \rangle$$

$$\begin{aligned} Init \triangleq\ & \land\ state = [p \in Proc \mapsto InitState] \\ & \land\ pending = [p \in Proc \mapsto \langle\,\rangle] \\ & \land\ rcvd = [p \in Proc \mapsto \langle\,\rangle] \end{aligned}$$

$IssueCmd(p,\ c) \triangleq$

The user issues request c to process p.

$$IssueCmd(p,\ c) \triangleq$$

$$\land\ Request(int,\ int',\ c,\ p)$$

$$IssueCmd(p,\ c) \triangleq$$
$$\land\ Request(int,\ int',\ c,\ p)$$
$$\land\ rcvd' = [rcvd\ \texttt{EXCEPT}\ ![p] = Append(@,\ c)]$$

$$IssueCmd(p,\ c) \triangleq$$

$$\land\ Request(int,\ int',\ c,\ p)$$

$$\land\ rcvd' = [rcvd\ \text{EXCEPT}\ ![p] = Append(@,\ c)]$$

$$\land\ \text{UNCHANGED}\ \langle state,\ pending \rangle$$

$SequenceCmd(p) \triangleq$

Move command from head of $rcvd[p]$ to tail of $pending[q]$, for all q.

$SequenceCmd(p) \triangleq \ldots$

See the handout.

$IssueReply(p) \triangleq$

p issues output for the command at the head of $pending[p]$.

$IssueReply(p) \triangleq \ldots$

See the handout.

$Next \triangleq \exists\, p \in Proc :$

$$Next \triangleq \exists\, p \in Proc : \lor \exists\, c \in Cmd : IssueCmd(p,\, c)$$

$$Next \triangleq \exists\, p \in Proc : \lor \exists\, c \in Cmd : IssueCmd(p,\, c)$$
$$\lor\ SequenceCmd(p)$$

$$
\begin{aligned}
Next \triangleq\ & \exists\, p \in Proc : \lor\ \exists\, c \in Cmd : IssueCmd(p,\, c) \\
& \lor\ SequenceCmd(p) \\
& \lor\ IssueReply(p)
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ & \exists\, p \in Proc : \lor \exists\, c \in Cmd : IssueCmd(p,\, c) \\
& \lor SequenceCmd(p) \\
& \lor IssueReply(p)
\end{aligned}
$$

$$
\begin{aligned}
ISpec \triangleq\ & \land Init \\
& \land \Box[Next]_{vars}
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ & \exists\, p \in Proc : \lor \exists\, c \in Cmd : IssueCmd(p,\, c) \\
& \lor SequenceCmd(p) \\
& \lor IssueReply(p)
\end{aligned}
$$

$$
\begin{aligned}
ISpec \triangleq\ & \land Init \\
& \land \Box[Next]_{vars} \\
& \land \forall\, p \in Proc : \land\ WF_{vars}(SequenceCmd(p))
\end{aligned}
$$

Every command in $rcvd[p]$ is eventually sequenced.

$$
\begin{array}{lcl}
Next & \triangleq & \exists\, p \in Proc : \lor \exists\, c \in Cmd : IssueCmd(p,\, c) \\
& & \qquad\qquad \lor\ SequenceCmd(p) \\
& & \qquad\qquad \lor\ IssueReply(p) \\
\\
ISpec & \triangleq & \land\ Init \\
& & \land\ \Box[Next]_{vars} \\
& & \land\ \forall\, p \in Proc : \land\ WF_{vars}(SequenceCmd(p)) \\
& & \qquad\qquad\qquad\ \land\ WF_{vars}(IssueReply(p))
\end{array}
$$

Every command in $pending[p]$ is eventually executed.

A Mutual Exclusion State Machine

Request: A record with two fields:

A Mutual Exclusion State Machine

Request: A record with two fields:

cmd in {"acquire", "release"}

A Mutual Exclusion State Machine

Request: A record with two fields:

cmd in {"acquire", "release"}

$proc$ in $Proc$

A Mutual Exclusion State Machine

Request: A record with two fields:

cmd in {"acquire", "release"}

$proc$ in $Proc$

Reply: The process acquiring the resource, or $None$ if no new owner.

Assumptions:

Assumptions:

- A process will not issue an $acquire$ if it is the owner or is waiting.

Assumptions:

- A process will not issue an $acquire$ if it is the owner or is waiting.
- Only the owner will issue a $release$.

Assumptions:

- A process will not issue an $acquire$ if it is the owner or is waiting.
- Only the owner will issue a $release$.

Avoids the need for an error response, and slightly simplifies $NewState$ and $Output$.

The state is a record with two fields:

VARIABLE int

CONSTANTS $Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$

Retain abstract interface.

MODULE *MutexSM*

VARIABLE *int*

CONSTANTS *Request*(_, _, _, _), *Reply*(_, _, _, _),
Proc

$$MInitState \triangleq [free \mapsto \text{TRUE},\ waiting \mapsto \langle\,\rangle]$$

$MNewState(s, c) \triangleq$

$MNewState(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN

ELSE

$MNewState(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN IF $s.free$

THEN

ELSE

ELSE

$MNewState(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN IF $s.free$

THEN $[s$ EXCEPT $!.free =$ FALSE$]$

ELSE

ELSE

Records are functions: $r.foo \triangleq r[\text{"foo"}]$

$MNewState(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN IF $s.free$

THEN $[s$ EXCEPT $!.free =$ FALSE$]$

ELSE

ELSE

$[r$ EXCEPT $!.foo = \ldots] \triangleq [r$ EXCEPT $![$"foo"$] = \ldots]$

$MNewState(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN IF $s.free$

THEN $[s$ EXCEPT $!.free = \text{FALSE}]$

ELSE $[s$ EXCEPT $!.waiting = Append(@, c.proc)]$

ELSE

$MNewState(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN IF $s.free$

THEN $[s$ EXCEPT $!.free = \text{FALSE}]$

ELSE $[s$ EXCEPT $!.waiting = Append(@, c.proc)]$

ELSE IF $s.waiting = \langle\rangle$

THEN

ELSE

$MNewState(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN IF $s.free$

THEN $[s$ EXCEPT $!.free =$ FALSE$]$

ELSE $[s$ EXCEPT $!.waiting = Append(@, c.proc)]$

ELSE IF $s.waiting = \langle\rangle$

THEN $[s$ EXCEPT $!.free =$ TRUE$]$

ELSE

$MNewState(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN IF $s.free$

THEN $[s$ EXCEPT $!.free =$ FALSE$]$

ELSE $[s$ EXCEPT $!.waiting = Append(@, c.proc)]$

ELSE IF $s.waiting = \langle\rangle$

THEN $[s$ EXCEPT $!.free =$ TRUE$]$

ELSE $[s$ EXCEPT $!.waiting = Tail(@)]$

$None \triangleq \text{CHOOSE } v : v \notin Proc$

$None \triangleq$ CHOOSE $v : v \notin Proc$

$MOutput(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN

ELSE

$None \triangleq$ CHOOSE $v : v \notin Proc$

$MOutput(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN IF $s.free$ THEN $c.proc$

ELSE $None$

ELSE

$None \triangleq$ CHOOSE $v : v \notin Proc$

$MOutput(s, c) \triangleq$

IF $c.cmd =$ "acquire"

THEN IF $s.free$ THEN $c.proc$

ELSE $None$

ELSE IF $s.waiting \neq \langle\,\rangle$ THEN $Head(s.waiting)$

ELSE $None$

$MCmd \triangleq [cmd : \{\text{“acquire”}, \text{“release”}\}, proc : Proc]$

$[a : S, b : T] \triangleq \{[a \mapsto s, b \mapsto t] : s \in S, t \in T\}$

INSTANCE $StateMachine$

WITH $InitState \leftarrow MInitState$
$NewState \leftarrow MNewState$
$Output \leftarrow MOutput$
$Cmd \leftarrow MCmd$

Defines $Spec$ to have definition from $StateMachine$, except with indicated substitutions.

$M\,InitState \triangleq \ldots$

$M\,NewState \triangleq \ldots$

$M\,Output \triangleq \ldots$

$M\,Cmd \triangleq \ldots$

INSTANCE $StateMachine$

WITH $InitState \leftarrow M\,InitState$

$NewState \leftarrow M\,NewState$

$Output \leftarrow M\,Output$

$Cmd \leftarrow M\,Cmd$

$InitState \triangleq \ldots$

$NewState \triangleq \ldots$

$Output \triangleq \ldots$

$Cmd \triangleq \ldots$

INSTANCE $StateMachine$

WITH $InitState \leftarrow InitState$

$NewState \leftarrow NewState$

$Output \leftarrow Output$

$Cmd \leftarrow Cmd$

$InitState \triangleq \ldots$

$NewState \triangleq \ldots$

$Output \triangleq \ldots$

$Cmd \triangleq \ldots$

INSTANCE $StateMachine$

Implicit instantiation by symbols of same name.

$InitState \triangleq \ldots$

$NewState \triangleq \ldots$

$Output \triangleq \ldots$

$Cmd \triangleq \ldots$

INSTANCE $StateMachine$

Already did this with int, $Request$, and $Reply$.

Specifying the Assumptions

Specifying the Assumptions

Assumptions:

- A process will not issue an $acquire$ if it is the owner or is waiting.
- Only the owner will issue a $release$.

Specifies the mutex's user/environment.

MODULE *MutexUser*

VARIABLE int

CONSTANTS $Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$,
$Proc$

MODULE *MutexUser*

VARIABLE int

CONSTANTS $Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$,
$Proc$

MODULE *Inner*

VARIABLE $state$

Use submodule instead of separate module
for inner spec.

MODULE *MutexUser*

VARIABLE int

CONSTANTS $Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$,
$Proc$

MODULE *Inner*

VARIABLE $state$

$state[p] \in \{$ “idle”, “waiting”, “owner” $\}$

$None \triangleq \text{CHOOSE } v : v \notin Proc$

$None \triangleq \text{CHOOSE } v : v \notin Proc$

$Init \triangleq state = [p \in Proc \mapsto \text{“idle”}]$

$IssueAcq(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ Request(int, int', [cmd \mapsto \text{“acquire”}, proc \mapsto p], p)$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$IssueAcq(p) \triangleq$

$\land\ state[p] = \text{“idle”}$

$\land\ Request(int, int', [cmd \mapsto \text{“acquire”}, proc \mapsto p], p)$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“waiting”}]$

$IssueRel(p) \triangleq$

$\land\ state[p] = \text{“owner”}$

$\land\ Request(int, int' [cmd \mapsto \text{“release”}, proc \mapsto p], p)$

$\land\ state' = [state \text{ EXCEPT } ![p] = \text{“idle”}]$

$$
\begin{array}{l}
RcvReply(p) \triangleq \\
\quad \exists\, r \in Proc \cup \{None\} : \\
\qquad \land\ Reply(int,\, int',\, r,\, p) \\
\qquad \land\ state' = \text{IF } r = p \\
\qquad\qquad\qquad \text{THEN } [state \text{ EXCEPT } ![p] = \text{“owner”}] \\
\qquad\qquad\qquad \text{ELSE } state
\end{array}
$$

The user receives a reply.

$RcvReply(p) \triangleq$

$\exists\, r \in Proc \cup \{None\} :$

$\land\ Reply(int,\ int',\ r,\ p)$

$\land\ state' = \text{IF}\ \ r = p$
THEN $[state$ EXCEPT $![p] =$ "owner"$]$
ELSE $state$

Always enabled. The system determines when a reply can be sent.

$$Next \triangleq \exists\, p \in Proc : IssueAcq(p) \lor IssueRel(p) \lor RcvReply(p)$$

$$ISpec \triangleq Init \land \Box[Next]_{\langle int, state \rangle}$$

$$Next \triangleq \exists\, p \in Proc : IssueAcq(p) \lor IssueRel(p) \lor RcvReply(p)$$

$$ISpec \triangleq Init \land \Box[Next]_{\langle int, state \rangle}$$

$$Next \triangleq \exists\, p \in Proc : IssueAcq(p) \lor IssueRel(p) \lor RcvReply(p)$$

$$ISpec \triangleq Init \land \Box[Next]_{\langle int, state \rangle}$$

$$I(state) \triangleq \text{INSTANCE } Inner$$

$$Spec \triangleq \exists\!\!\exists\, state : I(state)!ISpec$$

Combining User and System Specs

MODULE *MutexSystem*

VARIABLE int

CONSTANTS $Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$,
$Proc$

MODULE *MutexSystem*

VARIABLE *int*

CONSTANTS *Request*(_, _, _, _), *Reply*(_, _, _, _),
Proc

Env $\triangleq$ INSTANCE *MutexUser*

Sys $\triangleq$ INSTANCE *MutexSM*

MODULE *MutexSystem*

VARIABLE *int*

CONSTANTS $Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$, *Proc*

$Env \triangleq$ INSTANCE *MutexUser*

$Sys \triangleq$ INSTANCE *MutexSM*

$ClosedSystem \triangleq Env!Spec \land Sys!Spec$

Both the environment and the system behave correctly.

MODULE *MutexSystem*

VARIABLE *int*

CONSTANTS $Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$,
Proc

$Env \triangleq$ INSTANCE *MutexUser*

$Sys \triangleq$ INSTANCE *MutexSM*

$ClosedSystem \triangleq Env!Spec \land Sys!Spec$

Composition is conjunction.

MODULE *MutexSystem*

VARIABLE *int*

CONSTANTS $Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$,
Proc

$Env \triangleq$ INSTANCE *MutexUser*

$Sys \triangleq$ INSTANCE *MutexSM*

$ClosedSystem \triangleq Env!Spec \wedge Sys!Spec$

$OpenSystem \triangleq Env!Spec \Rightarrow Sys!Spec$

If the environment behaves correctly
then so does the system.

MODULE *MutexSystem*

VARIABLE *int*

CONSTANTS $Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$,
Proc

$Env \triangleq$ INSTANCE *MutexUser*

$Sys \triangleq$ INSTANCE *MutexSM*

$ClosedSystem \triangleq Env\,!\,Spec \wedge Sys\,!\,Spec$

$OpenSystem \triangleq Env\,!\,Spec \xrightarrow{+} Sys\,!\,Spec$

Disallows behaviors in which the system screws up and then the environment does too.

MODULE *MutexSystem*

VARIABLE int

CONSTANTS $Request(\_, \_, \_, \_)$, $Reply(\_, \_, \_, \_)$, $Proc$

$Env \triangleq$ INSTANCE $MutexUser$

$Sys \triangleq$ INSTANCE $MutexSM$

$ClosedSystem \triangleq Env!Spec \wedge Sys!Spec$

$OpenSystem \triangleq Env!Spec \stackrel{+}{\rightarrow} Sys!Spec$

That's all, folks!

See handout for URL.