Lecture 14 notes—Formal concurrency
Butler Lampson
6.826
October 20, 2020

Agenda for today

Goal: understand how to prove a concurrent program implements a spec.

State machines and TLA for concurrency, vs. languages.
Easy concurrency: making large atomic actions out of small ones
Examples of concurrency, both easy and hard

Reading question: What are the labels in PlusCal for? What goes wrong if you have too few labels? If you have too many?
State machine review

• Model any system as a global state with atomic transitions or steps. Some of the state is visible or external. The rest is internal. This god’s eye view works even if no agent can see the whole state.

• A trace, behavior, or history is a sequence of states:
  \[ s_0 \ s_1 \ \ldots \ s_n \]

• An action is a set of possible steps.
  \[- x := x + 1 \text{ is the steps } x=0 \rightarrow x=1, \ x=1 \rightarrow x=2, \ \ldots, \ x=17 \rightarrow x=18, \ \ldots \]

• In TLA+ an action is a (state, next state) predicate:
  \[- x := x + 1 \text{ becomes the predicate } x' = x + 1. \]
  ♦ This is short for \[ s' = [s \ \text{EXCEPT} \! [x] = x + 1] \]
  (sometimes written \[ s' = s[x := x + 1] = s[x+1/x] \]; pronounce “/” as “for”)

• A spec is a set of visible traces: what the system can do.

• Code C satisfies spec S if C’s visible traces are a subset of S’s
  So the spec says what the code is allowed to show externally.
Language

Expressions and assignment, combined with operators: ; ⇒ else * □ var.
Semantics: Compose actions into a bigger action. (BLK(c) = c blocks.)

<table>
<thead>
<tr>
<th>Command c</th>
<th>Action $a_c$</th>
<th>PlusCal syntax/meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v := e$</td>
<td>$v' = e$</td>
<td>expressions and assignment</td>
</tr>
<tr>
<td></td>
<td>$\land (\forall w \text{ except } v \mid w' = w)$</td>
<td></td>
</tr>
<tr>
<td>$c_1; c_2$</td>
<td>$\exists s_i \mid c_1(s, s_i) \land c_2(s_i, s')$</td>
<td>sequential composition</td>
</tr>
<tr>
<td>$e \Rightarrow c_0$</td>
<td>$e \land c_0$</td>
<td>if/await: if $p$ then $c_0$ else block</td>
</tr>
<tr>
<td>$c_1 \text{ else } c_2$</td>
<td>$c_1 \lor (\text{BLK}(c_1) \land c_2)$</td>
<td>else: $c_1$ if not blocked, else $c_2$</td>
</tr>
<tr>
<td>$c_0 *$</td>
<td>$\text{CLOSURE}(c_0) \land \text{BLK}(c_0)$</td>
<td>while: repeat $c_0$ until it blocks</td>
</tr>
</tbody>
</table>

Non-deterministic commands

| $c_1 \square c_2$ | $c_1 \lor c_2$ | either/or: $c_1$ or $c_2$ |
| $\text{var } v$ | $\exists t \mid v' = t$ | with: choose an arbitrary $v$ |

if $e$ then $c_1$ (same as $\{e \Rightarrow c_1\}$ else $c_2$)
else $c_2$

while $e$ do $c'$ (same as $(e \Rightarrow c')^*$)

CLOSURE($e \land c'$) \land \neg e'$
Language: Weakest preconditions

\(\text{wp}(c, Q):\) the weakest \(P\) such that \(\{P\} c \{Q\}\); it tells you the most about \(c\). 
\(\{P\} c \{Q\} \iff P \Rightarrow \text{wp}(c, Q).\) 
\(\{\text{wp}(c, Q)\} c \{Q\}.\) \(\text{wp}(c, Q) \land a_c \Rightarrow Q.\)

**Command** \(c\)  \quad **Action** \(a_c\)  \quad **wp**(\(c, Q)\) =

- \(v := e\)  \quad \(v' = e\)  \quad \(Q[v := e]\)  
  - What \(Q\) says about \(v\) is true of \(e\).
- \(c_1; c_2\)  \quad \(\exists s_i \mid c_1(s, s_i) \land c_2(s_i, s')\)  \quad \(\text{wp}(c_1, \text{wp}(c_2, Q))\)
- \(e \Rightarrow c_0\)  \quad \(e \land c_0\)  \quad \(\neg e \lor \text{wp}(c_0, Q)\)
- \(c_1 \text{ else } c_2\)  \quad \(c_1 \lor (\text{BLK}(c_1) \land c_2)\)  \quad \(\text{wp}(c_1, Q)\)  
  - \(\land (\text{BLK}(c_1) \Rightarrow \text{wp}(c_2, Q))\)
- \(c_0^*\)  \quad \(\text{CLOSURE}(c_0) \land \text{BLK}(c_0)\)  \quad \(\neg \text{BLK}(c_0) \Rightarrow \text{wp}(c_0, \text{wp}(c_0, Q))\)  
  - \(\land \text{BLK}(c_0) \Rightarrow Q\)

**Non-deterministic commands**

- \(c_1 \square c_2\)  \quad \(c_1 \lor c_2\)  \quad \(\text{wp}(c_1, Q) \land \text{wp}(c_2, Q)\)
- \(\text{var } v\)  \quad \(\exists t \mid v' = t\)  \quad \(\forall v \mid Q\)
State machines vs. languages

State machines are flat, except when you introduce an abstraction. Languages are recursive: build up the program from smaller parts.

State machines are foundational: you can express *any* system using only set theory and first order logic.

- There’s no built-in notion of sequential execution such as threads.
- You must build whatever you need (usually it’s easy; math is powerful)

Language semantics depends on non-interference: the build-up uses the facts that one command establishes to reason about the next one.

Proofs: state machines by an invariant, languages by weakest preconditions.
Concurrency and threads

Most generally,

- a state machine has a set of actions,
- zero or more of them are enabled (not blocked), and
- the next step is one of these actions

Any enabled action must maintain the invariant.
Sequential reasoning is simpler: only one next step.

A thread (or process) $h$ has a PC and a set of labeled actions of the form

$$pc[h] = l \Rightarrow a_l \land pc'[h] = l'$$

An action at $l$ in thread $h$

- is enabled only when $pc[h] = l$ (and $a_l$ is enabled too), and
- leaves the PC at the next action $l'$.

The next step can be from any thread whose PC is at an enabled action.

Here $a$ is an atomic action, one that runs as a single step.
We want big atomic actions. How?
Defining a state machine

A state machine is just a set of traces.
A set is defined by a predicate that’s true of its members.
So the state machine $S$ is defined by a predicate on its traces:

$$S = \text{Init}_S \land \square \text{Next}_S$$

*Init* is a state predicate that defines the set of initial states.
*Next* is an action (two-state) predicate that defines the possible steps

Typically $\text{Next} = a_1 \lor a_2 \lor \ldots \lor a_n$; each $a_i$ defines one of the actions.

$P$ is true of a trace if it’s true of the first state.
$A$ is true of a trace if it’s true of the pre and post states of the first step.

$\square Q$ is true of a trace if it’s true of every suffix; pronounce it “henceforth”.

So $\square A$ is true of a trace if $A$ is true of every step.

*C implements* $S$ if $C \Rightarrow S$: $\text{Init}_C \land \square \text{Next}_C \Rightarrow \text{Init}_S \land \square \text{Next}_S$
Reasoning about traces

Prove an invariant, a predicate $I$ true of every state in a trace; i.e., $\square I$.

Any set of states that includes all reachable states (predicate = set of states)
Strengthen it (remove unreachable states) to be inductive—to show $\square I$:

- $Init \Rightarrow I$  
  $I$ is true initially
- $I \land Next \Rightarrow I'$  
  every step preserves $I$

Then $Init \land \square Next \Rightarrow \square I$ follows by induction.

$I$ should be strong enough to tell you everything you want to know.

Often it’s much more complex than the invariant you need for the spec
Procedures and invariants

For a procedure $P$ with pre- and post-conditions $pre$ and $post$ that terminates in a state $done$, we want a generalized loop invariant, an $I$ for which

- $pre \Rightarrow I$ the precondition implies $I$
- $I \land done \Rightarrow post$ $I$ implies the postcondition when done

A call $[\alpha]P(x)[\beta]$ establishes the invariant $I_{post} \equiv (pc(th) = \beta) \Rightarrow post$

Any concurrent action enabled when $pc(th) = \beta$ must preserve $I_{post}$
Likewise for $I_{pre} \equiv (pc(th) = \alpha) \Rightarrow pre$
Data refinement

A state maps variable names to values.
Ex: If code $C$ has variables $cx, cy$ whose values are 5-bit strings, one state of $C$ is $c_0 = [cx := 01100, cy := 10010]$

A refinement mapping $m$ maps a state $c$ of $C$ to a state $s$ of $S$.
Ex: If $S$ variables $x, y$ are Nats, $m(c_0) = [x := 12, y := 18]$

$m_t$ works for a trace $t$ or step $cc$ (short trace) by applying it to each state:
$$t_S = m_t(t_C) = t_C \circ m$$

$C$ refines $S$ under $m$ if $m$ maps every trace of $C$ to a trace of $S$.
$$t_C \in C \Rightarrow m_t(t_C) \in S$$

$m([cx:=01100; cy:=10010]; [cx:=01100; cy:=00110]; [cx:=00110; cy:=00110])$
$$= [ x:=12; y:=18]; [x :=12; y:=6]; [x:=6; y:=6]$$
Logic for refinement

If $I$ is an $S$ predicate, $I^m = m \circ I$ is a $C$ predicate saying the “same” thing: $I^m(c) = I(m(c))$. $I^m$ goes backward:

- $m$ is $C \to S$, $I^m$ is $\text{set } S \to \text{set } C$, or $(S \to \text{Bool}) \to (C \to \text{Bool})$.

If $I$ is the logical formula for $I$, as a formula on $C$, $I^m$ is $I$ with each occurrence of a variable $v$ of $S$ replaced by $m_v(c)$.

- $m_v = m \circ \pi_v$ is just the part of $m$ that gives $v$’s value, where $\pi_v$ projects $v$—it maps a state $s$ to $v$’s value in $s$. So $m_x(c_0) = 12$ in $S$. 

```
s \xrightarrow{a} s' \\
m \uparrow \hspace{1cm} \downarrow a^m \hspace{1cm} m \uparrow \\
c \xrightarrow{m} c' 
```

```
s \xrightarrow{I} T/F \\
m \uparrow \hspace{1cm} \downarrow I^m \hspace{1cm} = \hspace{1cm} \\
c \xrightarrow{m} T/F 
```
Refining actions and traces

If $a$ is an $S$ action $a(s, s')$, $a^m(c, c') = a(m(c), m(c'))$ is a $C$ action that does the “same” thing. Like $I^m$, as a formula on $C$ actions, $a^m$ is $a$ with each $v$ replaced by $m_v(c)$.

So if $S$ is defined by the formula $S = \text{Init}_S \land \Box \text{Next}_S$, the refinement $S^m$ is defined by $S_m = S$ with each $v$ replaced by $m_v(c)$.

$C$ implements $S$ under $m$ iff $C \Rightarrow S^m$.

That was data refinement. Step refinement means that it’s always OK to take a stuttering step $\text{UNCHANGED}(v_1, \ldots, v_n)$. 
Atomic actions

What makes an action atomic?

- Host: The underlying execution model says so. Example: hardware makes load or store of a single word, or test and set atomic.

- Composition: It’s two steps $a_1; a_2$, and one of them commutes with every action $b$ in a different thread that’s enabled after $a_1$.
  
  $a$ and $b$ commute if $a; b = b; a$. This means that
  
  $a_1; b; a_2 = b; a_1; a_2$ or
  
  $a_1; b; a_2 = a_1; a_2; b$

  Either way, $a_1; a_2$ runs with no intervening step, so it’s atomic.

Host example: If $x, y, z$ are variables shared between threads, $x := y + z$ is not atomic on most hardware hosts, because other threads can change $y$ or $z$ in the middle. There are four host-atomic actions (machine instructions):

$$r_1 := y; r_2 := z; r_3 := r_1 + r_2; x := r_3$$
**Commuting**

Really easy case 1: If $a$ and $b$ share no variables that change, they commute. In distributed systems, this is **sharding** (partitioning, striping).

Really easy case 2: Producer-consumer: *put* and *get* for a buffer commute. *get* might block waiting for a *put*, so they must be in different threads. This is **streaming** or dataflow.

Easy case: $a$ and $b$ hold locks that conflict.

Easy case to *use*: **abstraction**—prove that (the code for) an action is atomic.

Hard: Anything else. You can do a proof or have a bug.

Eventual: Relax the spec.
Locks/mutexes

If $a$ and $c$ don’t commute, their threads must hold mutually **exclusive** locks. This guarantees that $a; c$ can’t happen, because $c$ is blocked.

What about lock (mutex) acquire and release, $m.\ acq$ and $m.\ rel$?

They only touch $m$, so commute with everything except $m$ actions.

When do two $m$ actions commute? What sequences can happen?

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a [\beta]$ $c$</td>
<td>Possible sequence ($c$ is enabled at $\beta$)?</td>
</tr>
<tr>
<td>1 $m.\ acq(h)$ $m.\ acq(h')$</td>
<td>No: $c$ is blocked by $h$ holding $m$</td>
</tr>
<tr>
<td>2 $m.\ acq(h)$ $m.\ rel(h')$</td>
<td>No: $c$ is blocked because $h'$ doesn’t hold $m$</td>
</tr>
<tr>
<td>3 $m.\ rel(h)$ $m.\ acq(h')$</td>
<td>OK</td>
</tr>
<tr>
<td>4 $m.\ rel(h)$ $m.\ rel(h')$</td>
<td>No: both threads can’t hold $m$, so one won’t do rel</td>
</tr>
</tbody>
</table>

• So $m.\ acq$ commutes with any $c$ at $\beta$
  
  – After $m.\ acq$ any $m$ action by $h'$ at $\beta$ is blocked (1,2).

• But $m.\ rel(h)$ doesn’t commute with $m.\ acq(h')$:
  
  – $m.\ rel(h); m.\ acq(h')$ is OK (3), but $m.\ acq(h'); m.\ rel(h)$ isn’t (2).

Can’t flip *every* $c$ before $rel$ to change $a; c; b$ into $c; a; b$, making $a; b$ atomic.
Definition of “commutes”

“c is enabled at β and commutes with a” is a; ([β] c) ⊆ c; a.
Semicolon means an $s_i$, so c commutes with a iff (with $u[a]u'$ for $a(u, u')$):

\[
\forall u, u' \mid (\exists u_i \mid u[a]u_i \land u_i[c]u' \land u_i(h.pc) = \beta) \quad \Rightarrow \quad (\exists s_i \mid u[c]s_i \land s_i[a]u')
\]

\[
\begin{array}{c}
u = s \xrightarrow{c} s_i \xrightarrow{a} u' = s' \\
\end{array}
\]

Anything $a; c$ does, $c; a$ also does. (But not vice-versa: if $a$ holds $m$ and $c$ does $m.acq$, there’s a $c; a$ step but no $a; c$ step.)
Simulation proof

We want to prove that atomic $a; [\beta] b \subseteq a; b$.

We make $a$ simulate $\text{skip}$ (the relation $=$) and $b$ simulate $a; b$, since we know more about $a$ than about $b$; every other command $c$ simulates itself.

$S$
\[
\begin{align*}
S & \xrightarrow{s \text{ skip}} s \xrightarrow{a; b} s' \\
U & \xrightarrow{u=s} u_i = s_i \xrightarrow{(pc=\beta)} u' = s'
\end{align*}
\]

$\vdash a; [\beta] b \subseteq a; b$

$S$
\[
\begin{align*}
S & \xrightarrow{s \text{ skip}} s \xrightarrow{s_c} s' \\
U & \xrightarrow{u=s} u_i = s_i \xrightarrow{(pc=\beta)} u_c \xrightarrow{(pc=\beta)} u' = s'
\end{align*}
\]

$\vdash a; c; [\beta] b \subseteq c; a; b$
Abstraction relation

So we make the AR ~ the identity except at \( \beta \), where it relates any state \( u_i \) for which \( s \rightarrow u_i \) to \( s \). So at \( \beta \) we haven’t yet done \( a \) in \( S \), but we have done \( a \) in \( U \). (Not just a function, since \( a \) may take many states to \( u_i \)):

\[
s \sim u \overset{\text{def}}{=} (u("h.pc") \neq \beta \land s = u) \lor (u("h.pc") = \beta \land s \boxed{a} u)
\]

Why is this an AR for \( u \rightarrow u' \)? Trivial if \( pc \neq \beta \) for both, since it’s =. From \( \beta \) we have either \( b \) or some \( c \) that commutes with \( a \).

<table>
<thead>
<tr>
<th>( S )</th>
<th>( \overset{\text{skip}}{\rightarrow} )</th>
<th>( \overset{a; b}{\rightarrow} )</th>
<th>( \overset{c}{\rightarrow} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( U )</td>
<td>( s )</td>
<td>( a )</td>
<td>( u )</td>
</tr>
<tr>
<td>( u = s \rightarrow u_i )</td>
<td>( \overset{a}{\rightarrow} )</td>
<td>( \overset{b}{\rightarrow} )</td>
<td>( \overset{c}{\rightarrow} )</td>
</tr>
<tr>
<td>( (pc = \beta) )</td>
<td>( (pc = \beta) )</td>
<td>( (pc = \beta) )</td>
<td>( (pc = \beta) )</td>
</tr>
</tbody>
</table>

If \( pc := \beta \), did \( a \). \( s \boxed{a} u_i \), so \( s \sim u_i \).

From \( pc = \beta \), if \( U \) does \( b \) need \( s \rightarrow s_i \rightarrow s' \), but \( u \) is this \( s_i \).

From \( pc = \beta \), if \( c \) is next need \( s_c \).
PlusCal for mutex

Here is the spec and a simple use to implement a critical section.

```plaintext
procedure acq(m) {l: await m = free; m := self; ret }
procedure rel(m) {l: if m = self then m := free else havoc; ret }
{ variable m = free;
  process(Proc ∈ 1..N)
    { ncs: skip; (* The Noncritical Section *)
      l1: acq(m)
      cs: skip; (* The Critical Section *)
      l2: rel(m); goto ncs  }
```

Here is code with less atomicity that uses a spin lock

```plaintext
procedure acq(m)
  variable t = held; { l1: while t ≠ free do {l2: t := m; m := held}; ret }
procedure rel(m) { l: m := free; ret }
```
Fast mutex

{ variables $x = 0; y = 0; b = [i \in 1..N \rightarrow \text{FALSE}]$; (* $b$ has one Boolean per process *)

process($Proc \in 1..N$) ; variable $j$;

{ $ncs$: skip; (* The Noncritical Section *)

start: $b[\text{self}]:= \text{TRUE}$;

$l1$: $x := \text{self}$;

$l2$: if ($y \not= 0$){ $l3$: $b[\text{self}] := \text{FALSE}$;

$l4$: await $y = 0$; goto start }

$l5$: $y := \text{self}$;

assert $x = \text{self} \Rightarrow y \not= 0$

$\delta l6$: if ($x \not= \text{self}$) { $l7$: $b[\text{self}] := \text{FALSE}$; $j := 1$; (* wait for all $b$’s to be false *)

$l8$: while ($j \leq N$) { await $\neg b[j]$; $j := j + 1$};

assert $y = \text{self} \Rightarrow \forall j: \neg (pc[j] \in \{l5, l6, cs\})$

$\epsilon l9$: if $y \not= \text{self}$ { $l10$: await $y = 0$; goto start }

assert $y \not= 0 \land \forall p \not= \text{self}: ((\neg pc[p] = \text{cs}) \land (pc[p] \in \{l5, l6\} \Rightarrow x \not= p))$

assert $\forall p \in 1..N \setminus \{\text{self}\} : pc[p] \not= \text{"cs"}$ (* mutual exclusion *)

cs: skip; (* The Critical Section *)

$l11$: $y := 0$

$l12$: $b[\text{self}] := \text{FALSE}$;

goto $ncs$ }

}
Backup

Symbolic execution?
Formulas vs. functions.
Models vs. reality.
State machines demand lots of “$x$ and $y$ commute” or “$x$ maintains $I$” arguments.
“Explicit yield” as a flexible strategy for bigger atomic actions (Armada).
PlusCal can do it by using fewer labels.
Bigger actions = fewer traces to reason about.
What about left movers? $acq$ is right mover only, $rel$ is left mover only.
Lock-protected ops are both-movers, because the lock ensures there can’t be any non-commuting ops to move over = all non-commuting ops are blocked.
Should give a concrete mover example.
# Language: Hoare triples

Taking a predicate $P$ as a function from a state $s$ to a Boolean, 
\[
\{P\} \ c \ \{Q\} \iff P(s) \land c \Rightarrow Q(s').
\]

<table>
<thead>
<tr>
<th>Command $c$</th>
<th>Action $a_c$</th>
<th>{(P)} (c) {(Q)} if</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v := e$</td>
<td>$v' = e$</td>
<td>$P = Q[v := e]$</td>
</tr>
<tr>
<td></td>
<td>$\land (\forall w \ \text{EXCEPT} \ v \mid w' = w)$</td>
<td></td>
</tr>
<tr>
<td>$c_1; c_2$</td>
<td>$\exists s_i \mid c_1(s, s_i) \land c_2(s_i, s')$</td>
<td>${P} \ c_1 \ {R} \ \text{and} \ {R} \ c_2 \ {Q}$</td>
</tr>
<tr>
<td>$e \Rightarrow c_0$</td>
<td>$e \land c_0$</td>
<td>$(P \Rightarrow \neg e) \lor {P} \ c_0 \ {Q}$</td>
</tr>
<tr>
<td>$c_1 \Leftrightarrow c_2$</td>
<td>$c_1 \lor (\text{BLK}(c_1) \land c_2)$</td>
<td>${P} \ c_1 \ {Q} \ \text{and}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>${P} \ c_2 \ {Q}$</td>
</tr>
<tr>
<td>$c_0 \ast$</td>
<td>\text{CLOSURE}(c_0) \land \text{BLK}(c_0)$</td>
<td>${P} \ c_0 \ {P} \land (P \land \text{BLK}(c_0) \Rightarrow Q)$</td>
</tr>
</tbody>
</table>

**Non-deterministic commands**

| $c_1 \bigtriangleup c_2$ | $c_1 \lor c_2$ | $\{P\} \ c_1 \ \{Q\} \ \text{and} \ \{P\} \ c_2 \ \{Q\}$ |
| $\text{var } v$ | $\exists t \mid v' = t$ | $P = \forall v \mid Q$ |
Language: Strongest postconditions

\( sp(c, P) \): the strongest \( Q \) such that \( \{ P \} \ c \ \{ Q \} \); it tells you the most about \( c \). This is symbolic execution.

\( \{ P \} \ c \ \{ Q \} \ \Leftrightarrow \ sp(c, P) \Rightarrow Q. \quad \{ P \} \ c \ \{ sp(c, P) \}. \quad P \land a_c \Rightarrow sp(c, P) \)

**Command** \( c \)   **Action** \( a_c \)   **sp** \( (c, P) = \)
\( v := e \)   \( v' = e \)   \( \exists t \mid v = e[v := t] \)
\( \land (\forall w \text{ EXCEPT } v \mid w' = w) \)   \( \land P[v := t] \)
\( c_1 ; c_2 \)   \( \exists s_i \mid c_1(s, s_i) \land c_2(s_i, s') \)   \( \text{sp}(c_2, \text{sp}(c_1, P)) \)
\( e \Rightarrow c_0 \)   \( e \land c_0 \)   \( \neg e \lor \text{sp}(c_0, P) \)
\( c_1 \boxdot c_2 \)   \( c_1 \lor (\text{BLK}(c_1) \land c_2) \)   \( \text{sp}(c_1, P) \lor (\text{BLK}(c_1) \Rightarrow \text{sp}(c_2, P)) \)
\( c_0 \ast \)   \( \text{CLOSURE}(c_0) \land \text{BLK}(c_0) \)   \( \text{sp}(c_0, \text{sp}(c_0, \neg \text{BLK}(c_0) \land P)) \lor \text{BLK}(c_0) \land P \)

**Non-deterministic commands**
\( c_1 \Box c_2 \)   \( c_1 \lor c_2 \)   \( \text{sp}(c_1, P) \lor \text{sp}(c_2, P) \)
\( \text{var } v \)   \( \exists t \mid v' = t \)   \( P \land \exists t \mid v = t \)