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:
  $s_0 \ s_1 \ \dots \ s_n$
- An **action** is a set of possible steps.
  - $x \coloneqq x + 1$ is the steps x=0 → x=1, x=1 → x=2, …, x=17 → x=18, …
- In TLA+ an action is a (state, next state) predicate:
  - $x := x + 1$ becomes the predicate $x' = x + 1$.
    - ♦ This is short for $s' = [s \text{ EXCEPT } ! [x] = x + 1]$
      (sometimes written $s' = s[x \coloneqq x + 1] = s[x{+}1/x]$; 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.)

Command c	**Action a_c**	**PlusCal syntax/Meaning**
$v \coloneqq e$	$v' = e$ $\land\ (\forall w$ EXCEPT $v \mid w'=w)$	expressions and assignment
$c_1; c_2$	$\exists s_i \mid c_1(s, s_i) \land c_2(s_i, s')$	sequential composition
$e \Rightarrow c_0$	$e \land c_0$	**if/await**: if p then c_0 else block
c_1 **else** c_2	$c_1 \lor (\text{BLK}(c_1) \land c_2)$	**else**: c_1 if not blocked, else c_2
$c_0\ *$	CLOSURE$(c_0) \land$ BLK(c_0)	**while**: repeat c_0 until it blocks
Non-deterministic commands		
$c_1 \square c_2$	$c_1 \lor c_2$	**either/or:** c_1 or c_2
var v	$\exists t \mid v' = t$	**with**: choose an arbitrary v
if e **then** c_1 **else** c_2	$(e \land c_1) \square (\neg e \land c_2)$	same as $\{e \Rightarrow c_1\}$ **else** c_2 or $\{e \Rightarrow c_1\} \square \{\neg p \Rightarrow c_2\}$
while e **do** c'	CLOSURE$(e \land c') \land \neg e'$	same as $(e \Rightarrow c')\ *$

Language: Weakest preconditions

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

Command c	**Action a_c**	$\boldsymbol{wp(c,Q)}$ **=**
$v \coloneqq e$	$v' = e$ $\land\ (\forall w\ \text{EXCEPT}\ v \mid w'=w)$	$Q[v \coloneqq e]$ What Q says about v is true of e.
$c_1; c_2$	$\exists s_i \mid c_1(s,s_i) \land c_2(s_i,s')$	$wp\big(c_1, wp(c_2,Q)\big)$
$e \Rightarrow c_0$	$e \land c_0$	$\neg e \lor wp(c_0,Q)$
c_1 **else** c_2	$c_1 \lor (\text{BLK}(c_1) \land c_2)$	$wp(c_1,Q)$ $\land\ \big(\text{BLK}(c_1) \Rightarrow wp(c_2,Q)\big)$
$c_0\ *$	$\text{CLOSURE}(c_0) \land \text{BLK}(c_0)$	$\neg\text{BLK}(c_0) \Rightarrow wp\big(c_0, wp(\mathrm{c}_0,Q)\big)$ $\land\ \text{BLK}(c_0) \Rightarrow Q$
Non-deterministic commands		
$c_1 \,\square\, c_2$	$c_1 \lor c_2$	$wp(c_1,Q) \land wp(c_2,Q)$
var v	$\exists t \mid v' = t$	$\forall v \mid Q$

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:

$$\boxed{\boldsymbol{S = Init_S \land \Box\, Next_S}}$$

$Init$ is a state predicate that defines the set of initial states.
$Next$ is an action (two-state) predicate that defines the possible steps
Typically $Next = a_1 \lor a_2 \lor \ldots \lor a_n$; 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.
$\Box Q$ is true of a trace if it's true of every suffix; pronounce it "henceforth".
So $\Box A$ is true of a trace if A is true of every step.

C **implements** S if $C \Rightarrow S$: $Init_C \land \Box\, Next_C \Rightarrow Init_S \land \Box\, Next_S$

Reasoning about traces

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

Any set of states that includes all reachable states (predicate = set of states)

Strengthen it (remove unreachable states) to be *inductive*—to show $\Box I$:

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

Then $Init \land \Box Next \Rightarrow \Box I$ 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
$I \wedge done \Rightarrow post$	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 cxc, y whose values are 5-bit strings, one state of C is

$$c_0 = [cx \coloneqq 01100, cy \coloneqq 10010]$$

A *refinement mapping* m maps a state c of C to a state s of S.

Ex: If S variables x, y are Nats, $m(c_0) = [x \coloneqq 12, y \coloneqq 18]$

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.

$$t_C \in C \Rightarrow m_t(t_C) \in S$$

$m([cx{\coloneqq}01100; cy{\coloneqq}10010]; [cx{\coloneqq}01100; cy{\coloneqq}00110]; [cx{\coloneqq}00110; cy{\coloneqq}00110])$

$= [\ x{\coloneqq}12; \qquad y{\coloneqq}18]; \qquad [x\ {\coloneqq}12; \qquad y{\coloneqq}6]; \qquad [x{\coloneqq}6; \qquad y{\coloneqq}6 \qquad]$

Logic for refinement

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

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

If $\boldsymbol{I}$ is the logical formula for I, as a formula on C, $\boldsymbol{I^m}$ is $\boldsymbol{I}$ 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 π_v *projects* v—it maps a state s to v’s value in s. So $m_x(c_0) = 12$ in S.

$$\begin{tikzcd}
s \arrow[r, Rightarrow, "a"] & s' \\
c \arrow[u, "m"] \arrow[r, Rightarrow, "a^m"] & c' \arrow[u, "m"]
\end{tikzcd}$$

$$\begin{tikzcd}
s \arrow[r, "I"] & T/F \\
c \arrow[u, "m"] \arrow[r, "I^m"] & T/F \arrow[u, equal, "="]
\end{tikzcd}$$

Refining actions and traces

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

So if S is defined by the formula $\boldsymbol{S = Init_S} \wedge \Box \boldsymbol{Next_S}$, the refinement S^m is defined by $\boldsymbol{S_m = S}$ with each v replaced by $m_v(c)$.

C implements S under m iff $\boldsymbol{C \Rightarrow S^m}$.

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

  $$a_1; b; a_2 = b; a_1; a_2 \text{ or}$$

  $$a_1; b; a_2 = a_1; a_2; b$$

  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, $x \coloneqq y + z$ 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):

$$r_1 \coloneqq y; r_2 \coloneqq z; r_3 \coloneqq r_1 + r_2; x \coloneqq r_3$$

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?

	$a\ [\beta]$	c	Possible sequence (c is enabled at β)?
1	$m.acq(h)$	$m.acq(h')$	No: c is blocked by h holding m
2	$m.acq(h)$	$m.rel(h')$	No: c is blocked because h' doesn’t hold m
3	$m.rel(h)$	$m.acq(h')$	OK
4	$m.rel(h)$	$m.rel(h')$	No: both threads can’t hold m, so one won’t do rel

- So $m.acq$ commutes with any c at β
  - After $m.acq$ any m action by h' at β 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 $a; ([\beta]\ c) \subseteq c; a$.
Semicolon means an s_i, so c commutes with a iff (with $u\boxed{a}u'$ for $a(u, u')$):

$$\forall\, u, u' \mid \quad (\,\exists u_i \mid u \boxed{a} u_i \wedge u_i \boxed{c} u' \wedge u_i(\text{h.pc}) = \beta\,) \qquad u \boxed{a; ([\beta]\ c)} u'$$

$$\Rightarrow (\,\exists s_i \mid u \boxed{c}\ s_i \wedge\ s_i \boxed{a} u'\,) \qquad u \quad \boxed{a; c} \quad u'$$

$$\begin{tikzcd}
u = s \arrow[r, Rightarrow, "c"] & s_i \arrow[r, Rightarrow, "a"] & u' = s' \\
u = s \arrow[u, "="] \arrow[r, Rightarrow, "a"] & \underset{(pc = \beta)}{u_i} \arrow[u, "?"] \arrow[r, Rightarrow, "c"] & u' = s' \arrow[u, "="]
\end{tikzcd}$$

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 **skip** (the relation $=$) and b simulate $a; b$, since we know more about a than about b; every other command c simulates itself.

$$\begin{tikzcd}
S & s \arrow[r, Rightarrow, "\mathbf{skip}"] \arrow[d, equal, "="'] & s \arrow[r, Rightarrow, "a; b"] \arrow[d, leftrightarrow, "\sim"'] & s' \arrow[d, equal, "="'] \\
U & u{=}s \arrow[r, Rightarrow, "a"] & \begin{array}{c} u_i{=}s_i \\ (pc{=}\beta) \end{array} \arrow[r, Rightarrow, "b"] & u'{=}s'
\end{tikzcd}$$

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

$$\begin{tikzcd}
s \arrow[r, Rightarrow, "\mathbf{skip}"] \arrow[d, equal, "="'] & s \arrow[r, Rightarrow, "c"] \arrow[d, leftrightarrow, "\sim"'] & s_c \arrow[r, Rightarrow, "a; b"] \arrow[d, "a"'] & s' \arrow[d, equal, "="'] \\
u{=}s \arrow[r, Rightarrow, "a"] & \begin{array}{c} u_i{=}s_i \\ (pc{=}\beta) \end{array} \arrow[r, Rightarrow, "c"] & \begin{array}{c} u_c \\ (pc{=}\beta) \end{array} \arrow[r, Rightarrow, "b"] & u'{=}s'
\end{tikzcd}$$

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

Abstraction relation

So we make the AR $\sim$ the identity except at β, where it relates any state u_i for which $s \rightarrow u_i$ to s. So at β 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):

$$s{\sim}u \overset{\text{def}}{=} (u(\text{"h.pc"}) \neq \beta \wedge s = u) \vee (u(\text{"h.pc"}) = \beta \wedge s \boxed{a} u)$$

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

S U	$\begin{tikzcd} s \arrow[r, Rightarrow, "\textbf{skip}"] \arrow[d, equal, "="'] & s \arrow[d, leftrightarrow, "a"'] \\ u = s \arrow[r, Rightarrow, "a"] & u_i \end{tikzcd}$ $(pc = \beta)$	$\begin{tikzcd} s \arrow[r, Rightarrow, "a; b"] \arrow[d, leftrightarrow, "a"'] & s' \arrow[d, equal, "="'] \\ u \arrow[r, Rightarrow, "b"] & u' \end{tikzcd}$ $(pc = \beta)$	$\begin{tikzcd} s \arrow[r, Rightarrow, "c"] \arrow[d, "a"'] & s_c \arrow[d, "a"'] \\ u_i{=}s_i \arrow[r, Rightarrow, "c"] & u_c \end{tikzcd}$ $(pc{=}\beta)$
	If $pc := \beta$, did a. $s \boxed{a} u_i$, so $s{\sim}u_i$.	From $pc = \beta$, if U does b need $s \overset{a}{\rightarrow} s_i \overset{b}{\rightarrow} s'$, 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.

```
procedure $acq(m)$ {$l$: await $m = free$; $m \coloneqq self$; ret };
procedure $rel(m)$ {$l$: if $m = self$ then $m \coloneqq free$ else havoc; ret };
{ variable $m = free$;
  process($Proc \in 1..N$)
      { $ncs$:  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

```
procedure $acq(m)$
      variable $t = held$; { $l1$: while $t \neq free$ do {$l2$: $t \coloneqq m$; $m \coloneqq held$}; ret }
procedure $rel(m)$ { $l$: $m \coloneqq free$; ret }
```

Fast mutex

```
{ variables $x = 0; y = 0; b = [i \in 1..N \mapsto \text{FALSE}]$;     (* $b$ has one Boolean per process *)
  process$(Proc \in 1..N)$ ; variable $j$;
      { $ncs$:    skip;  (* The Noncritical Section *)
        $start$: $b[self] := \text{TRUE}$;
          $l1$: $x := self$;
          $l2$: if $(y \neq 0)$ { $l3$: $b[self] := \text{FALSE}$;
                              $l4$: await $y = 0$;  goto $start$  };
          $l5$: $y := self$;
          assert $x = self \Rightarrow y \neq 0$
          $\delta$ $l6$: if $(x \neq self)$ {  $l7$: $b[self] := \text{FALSE}$; $j := 1$;    (* wait for all $b$'s to be false *)
                                     $l8$: while $(j \leq N)$ {await $\neg b[j]$; $j := j + 1$};
                                  assert $y = self \Rightarrow \forall j: \neg(pc[j] \in \{l5, l6, cs\})$
                                  $\epsilon$ $l9$: if $y \neq self$ { $l10$: await $y = 0$; goto $start$}    };
          assert $y \neq 0 \wedge \forall p \neq self: ((\neg pc[p] = cs) \wedge (pc[p] \in \{l5, l6\} \Rightarrow x \neq p))$
          assert $\forall p \in 1..N \setminus \{self\} : pc[p] \neq \text{"cs"}$    (* mutual exclusion *)
          $cs$:   skip; (*  The Critical Section *)
          $l11$: $y := 0$ ;
          $l12$: $b[self] := \text{FALSE}$ ;
                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,
$\{P\}\ c\ \{Q\} \Leftrightarrow P(s) \land c \Rightarrow Q(s')$.

Command c	**Action a_c**	**$\{P\}\ c\ \{Q\}$ if**
$v \coloneqq e$	$v' = e$ $\land\ (\forall w \text{ EXCEPT } v \mid w'=w)$	$P = Q[v \coloneqq e]$
$c_1; c_2$	$\exists s_i \mid c_1(s, s_i) \land c_2(s_i, s')$	$\{P\}\ c_1\ \{R\}$ and $\{R\}\ c_2\ \{Q\}$
$e \Rightarrow c_0$	$e \land c_0$	$(P \Rightarrow \neg e) \lor \{P\}\ c_0\ \{Q\}$
$c_1 \boxtimes c_2$	$c_1 \lor (\text{BLK}(c_1) \land c_2)$	$\{P\}\ c_1\ \{Q\}$ and $\{P \land \text{BLK}(c_1)\}\ c_2\ \{Q\}$
$c_0\ *$	$\text{CLOSURE}(c_0) \land \text{BLK}(c_0)$	$\{P\}\ c_0\ \{P\} \land (P \land \text{BLK}(c_0) \Rightarrow Q)$
Non-deterministic commands		
$c_1 \square c_2$	$c_1 \lor c_2$	$\{P\}\ c_1\ \{Q\}$ and $\{P\}\ c_2\ \{Q\}$
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*.

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

Command c	**Action a_c**	**$sp(c, P)$ =**
$v \coloneqq e$	$v' = e$ $\wedge\ (\forall w \text{ EXCEPT } v \mid w' = w)$	$\exists t \mid\ v = e[v \coloneqq t]$ $\wedge\ P[v \coloneqq t]$
$c_1; c_2$	$\exists s_i \mid c_1(s, s_i) \wedge c_2(s_i, s')$	$sp\big(c_2, sp(c_1, P)\big)$
$e \Rightarrow c_0$	$e \wedge c_0$	$\neg e \vee sp(c_0, P)$
$c_1 \boxtimes c_2$	$c_1 \vee (\text{BLK}(c_1) \wedge c_2)$	$sp(c_1, P)$ $\vee\ (\text{BLK}(c_1) \Rightarrow sp(c_2, P))$
$c_0\ *$	$\text{CLOSURE}(c_0) \wedge \text{BLK}(c_0)$	$sp(c_0, sp(c_0, \neg\text{BLK}(c_0) \wedge P))$ $\vee\ \text{BLK}(c_0) \wedge P$
Non-deterministic commands		
$c_1 \,\square\, c_2$	$c_1 \vee c_2$	$sp(c_1, P) \vee sp(c_2, P)$
var v	$\exists t \mid v' = t$	$P \wedge \exists t \mid v = t$