Class 9
6.852
Fall, '01

bring: chalk
code handout

Last time:

Introduced basic automaton model for reactive system components.
I/O automata.

Went over basic defs, examples based on simple non-FT consensus proc
reliable FIFO channels

states, start, sig, trans, tasks

+ defined what it means to compose I/O automata

Now finish up with: important properties composition satisfies
fairness
proof methods, esp. invariants
+ sim. relns

Then go on to asynch network algs.

<u>Basic composition results</u> (no liveness yet)

To show the model works right for describing system in pieces.

<u>Projection + pasting</u>:

<u>Thm 1</u>: (Projection)

1. If $\alpha \in execs(A)$ then $\underbrace{\alpha | A_i}$ $\in execs(A_i)$ for every i

   delete actions (+ following states) for non-actions of A_i

   + project on A_i's state.

2. If $\beta \in traces(A)$ then $\underbrace{\beta | A_i}$ $\in traces(A_i)$ ~~[illegible]~~ for every $i \in I$

   restr. to actions of A_i

<u>Thm 2</u>: (Pasting)

1. If $\alpha_i \in execs(A_i)$ for all i,

   β is a seq of actions in $ext(A)$ s.t. $\beta | A_i = trace(\alpha_i)$ for all i

   then $\exists$ exec α of A s.t. $\beta = trace(\alpha)$

   + $\alpha_i = \alpha | A_i$ for all $i \in I$.

2. If β is a seq. of actions in $ext(A)$ + $\beta | A_i \in traces(A_i)$ for all i then $\beta \in traces(A)$

Proofs LTTR. Allows compositional reasoning:

<u>Thm 3</u>: (Substitutivity)

Suppose A + A' have same ext. signature

B + B' " " " "

and suppose $\begin{cases} traces(A') \subseteq traces(A) \\ traces(B') \subseteq traces(B). \end{cases}$

Then $traces(A' \times B') \subseteq traces(A \times B)$

(a kind of "implementation relationship")

Pf: Project, then paste.

(It's also possible to state some ~~results about~~ substitutivity results for the case where we only assume the original implementation relationships hold ~~only in certain~~ under certain restrictions on the environment.
Left for HW.)

Hiding: $hide_{\Sigma}(A)$ = same as A but with Σ reclassified as internal

Σ = outputs

Renaming of actions

8.3 Fairness

To complete the basic model defs, need notion of fair exec for an IOA.
Want to say that each task (~~or~~ thread) keeps getting turns to do steps.
Well, no steps of that task might be enabled, but at least the task should get a chance "if it wants".

Formally, exec frag α defined to be fair if for all $C \in Tasks(A)$, one of the following holds:

1. α is ~~fin~~ finite + no action of C is enabled in final state of A
2. α is infinite + contains ∞ many steps with acts in C
3. " " " " " " " states occurrences in which C not enabled.

fair exec of A

fair trace of A = trace of fair exec of A.

Ex: Channel

first example exec I gave was fair - finite + nothing enabled at end.
2nd example not fair - leaves something enabled at end.
send send send ... infinite + not fair - doesn't give turns to the 1 task.

Ex: Process

Has separate tasks for {sending to each other process
output
Means must keep ~~trying to do~~ ~~doing~~ there forever.
trying to do

Fairness also behaves nicely w.r.t composition - results analogous to the unfair case.

Ex: Discuss fair execs of composition of procs + channels.
All msgs sent are delivered.
After init, send val forever (if changes send new one)
If get whole vector, output forever.

8.4. Properties + Proof Methods

To model inputs + outputs for problems we generally use input + output actions, instead of state vars. (as in the synch model)

Properties to prove:

Invariants: Properties of states, true in all reachable states
Prove by induction (sometimes in batches).
Step granularity is finer than rounds, so proofs get harder.

~~∃traces(A) ⊆ Q, some set of traces satisfy some property~~
~~or ∃fairtraces(A) ⊆ Q~~

~~One automaton A "implements" another automaton B, in the sense that traces(A) ⊆ traces(B).~~

<u>Trace properties</u>: Any <u>property of external behavior sequences</u> ~~traces~~ of an automaton.

Formally, P = (external signature, set of sequences of acts in the sig.)

in + out only

(sig(P), traces(P))

Property

Say <u>A satisfies P</u> if it has the same external signature +

$$\begin{cases} \text{traces}(A) \subseteq \text{traces}(P) \\ \text{or} \\ \text{fairtraces}(A) \subseteq \text{traces}(P) \end{cases}$$

<u>2 possible meanings</u>

Depending on <u>whether we're interested in liveness</u> or not.

<u>All the problems we'll deal with in asynch systems can be formulated as trace properties</u>.
+ we'll <u>usually be concerned about fairness</u>, so we'll use the second notion.

<u>Incidentally</u>, 2 types of trace properties are important enough to have special names:

<u>Safety properties</u>: traces(P) nonempty, prefix-closed, limit-closed
Can interpret as "something bad" never happens.

Examples (we'll see)

consensus: agreement, validity

set of seqs of init or decide actions in which we never get disagreement, or never get violation of validity.

graph searching: correct shortest path

mutual exclusion: no 2 simultaneous grants

So over why there are safety properties: $\begin{cases} \lambda \text{ doesn't violate} \\ \text{If trace OK then so are all prefixes} \\ \text{If all prefixes OK, trace OK.} \end{cases}$

(Generally can prove by relating to a state invariant and proving the invariant by induction.)

Liveness property P:

Every finite seq. over sig(P) has extension in traces(P).

Ex: Termination properties can be expressed in this way.

No matter where we are, we could still terminate in the future.

Curious fact: Every trace property can be expressed as a combination of a safety property + a liveness property.

Formally, if (sig, T) is a trace property, then

$\exists$ (sig, S) safety property

+ $\exists$ (sig, L) liveness property

s.t. $T = S \cap L$.

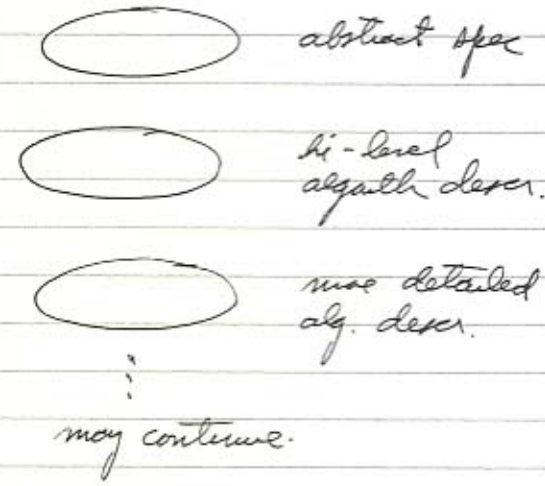
So it's no accident that in a problem spec, you see a bunch of safety props, then a bunch of liveness properties, + nothing else.

Liveness pfs less systematic than safety proofs.
Basic ideas: milestones, progress functions.

Hierarchical proofs

Very important strategy for proving correctness of complex, detailed systems.

Formulate in a series of levels, all automata



Can have high level centralized, lower levels distributed
" " inefficient but simple, " " optimized
" " with large granularity actions, " " finer granularity

In all these cases, lower levels harder to understand.
Try not to reason about them directly, but rather by relating them to higher-level algs.

Method similar to what I showed for synchronous algs:
optimized alg run side-by-side with unoptimized version
+ invariant proved relating the 2 states.

Invariant relating 2 states called simulation relation.
Showed using induction.
Want to take advantage of same powerful method for asynch systems.

But there are a few problems...
Much more freedom allowed in asynch model, in order of steps, + in new states.
∴ Harder to determine which executions to compare.

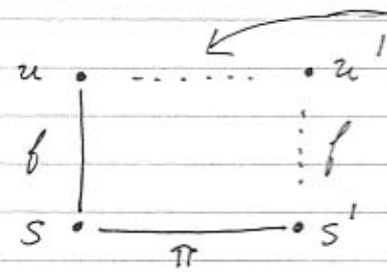
Turns out it's <u>enough to obtain a 1-way relationship</u> between execs, showing for each exec. of the low-level alg., there exists a corresponding exec. of higher-level alg.

<u>Def</u>: Assume A & B have same ext. sig.
f a binary relation over <u>states(A) and states(B)</u>
$\subseteq$ states(A) $\times$ states(B)
(Write $(s,u) \in f$ or $u \in f(s)$.)

<u>Then f is sim reln from A to B provided that</u>

1. If $s \in$ start(A) then $f(s) \cap$ start(B) $\neq \phi$
   ($\exists u \in$ start(B) $\cap f(s)$)
2. If s, u are reachable states of A and B resp., $u \in f(s)$, & $(s, \pi, s') \in$ trans(A), then $\exists$ exec frag α of B starting with u, and ending with some $u' \in f(s')$, with trace(α) = trace(π)

Explain:

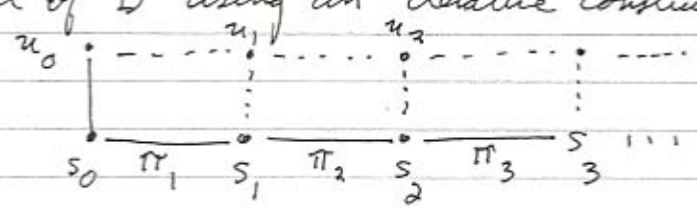


can be 0, 1 or more steps

π external $\rightarrow$ π + some internals
π internal $\rightarrow$ 0 or more internals

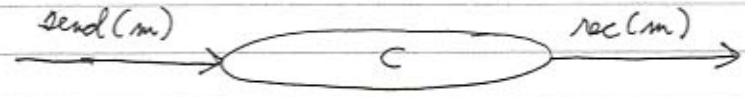
<u>Theorem</u>: If there's a sim reln from A to B, then $\underbrace{\text{traces(A)}}_{\text{all traces, not just finite}} \subseteq$ traces(B)

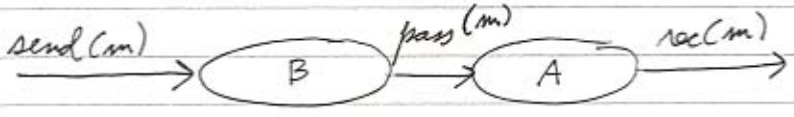
<u>Pf</u>: Fix an exec. of A, create a corresponding exec of B using an iterative construction:



Gives full corresp. exec.
Easy to see corresp.
Can be finite or infinite.

Ex: 2 channels implement one





renaming some actions to "pass"

Claim: $A \times B$ implements C, in sense of traces$(A) \subseteq$ traces(B).

Sim reln f: $(s, u) \in f$ if u.queue is concat of $s.A.$queue + $s.B.$queue

If it is ~~the~~ a sim. reln, proves implementation ~~the~~

How to prove: Check the 2 conds:

Start: Empty queues correspond ✓

Step: Describe step corresp:

Helps to give action corresp. first.

If $\pi = send(m)$, use $send(m)$ in C
(unique state generable from u in C)

If $\pi = rec(m)$, use $rec(m)$ in C ~~[illegible]~~

If $\pi = pass(m)$, use λ

Clearly, traces preserved.

Show correspondence:

Means we have to check enabling for high-level corresp. actions, + check correspondences for final states.

Breaks into cases:

π = send (m)

No enabling issues (input).
Must check $(s', u') \in f$ – new states.
Have $(s, u) \in f$, add same m to end of u. queue + s. B. queue,
so still correspond.

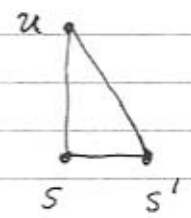
π = rec (m)

Check rec(m) for same m also enabled in u.
Know m is first on s. A. queue
By correspondence $((s, u) \in f)$, m is first on u. queue.
So enabled in u.

Effects: Removing heads of both preserves correspondence.

π = pass (m)

No enabling issues (no hi-level actions)
Just need to show state corresp.
$(s', u) \in f$



Know $(s, u) \in f$, thus, u. queue = concat of s. A. queue + s. B. queue
Concat doesn't change as a result of this step, so also
u. queue = concat. of s'. A. queue + s'. B. queue.

<u>Another example</u> — for HW

Consider new channel $\mathbb{D}$ with <u>duplicate</u> internal action that allows any msg to be duplicated consecutively in the queue.

$$x \; \underset{\substack{\uparrow \\ \text{a single msg}}}{m} \; y \xrightarrow{dup} x \; \underbrace{m \; m}_{\text{two copies}} \; y$$

But also modify by adding <u>alternating tag</u>, 0 or 1, to msgs (+ duplicating these along with msgs).

Also, <u>remember last tag delivered</u> + throw away ~~following copies with same tag~~ a msg that's first on the queue but has same tag as the one ~~previously~~ most recently delivered.

LTTR to formalize. Key idea of <u>Alternating Bit Protocol</u>

Claim this channel $\mathbb{D}$ implements ordinary reliable FIFO channel A, trace inclusion.

<u>Can show using sim. reln.:</u>

Ub. A. queue = collapsed version of s. $\mathbb{D}$. queue
(combine those with same tag into one, ignore first msg if same tag as last delivered + then erase tags)

<u>Proof</u>: LTTR.

send, rec $\xrightarrow{map}$ themselves

dup, throwaway $\longrightarrow \lambda$

Asynch Networks

Processes communicating via channels

can be { point-to-point
mcast
bcast

Quick sketch of model, then move to typical problems like
- leader election
- setting up search structures (BFS, etc.)
- MST

Compare results with synch networks.

Send/receive systems

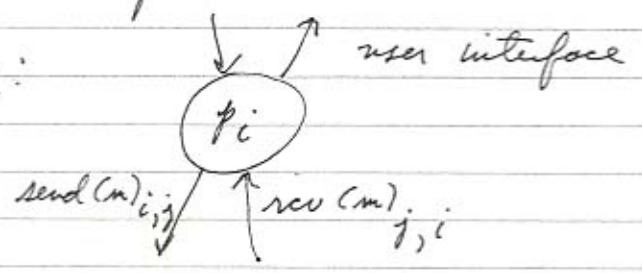
Use pt-pt channels.

Digraph $G = (V, E)$, process automata associated with nodes
channel automata " " directed edges

No rounds. Allow asynchrony in steps of all components.

Model both procs + channels as IOAs

Processes:



Problems to be solved get exposed as sets of (allowable) traces at user interface.

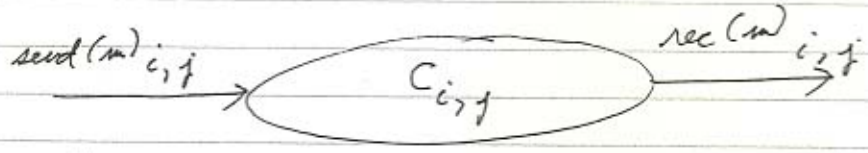
Modeling failures:

e.g. $\xrightarrow{stop_i}$ p_i input, disables all l.c. actions

Since in external interface, allows problems to be stated in terms of occurrences of failures.
(generally, liveness properties)

Channels:

Send / receive:



Can consider different "channel semantics" – different kinds of channels with this interface:

reliable FIFO

various kinds of less reliable:

allowing losses, duplicates, reordering, e.g.

(will come back to this later)

for now, just consider rel. FIFO

This is just the channel already described – state is queue, send adds, rec removes head

Liveness: all receives in 1 task.

alternative description of channel: $\underbrace{\text{Interface + trace constraints}}_{\text{trace property.}}$

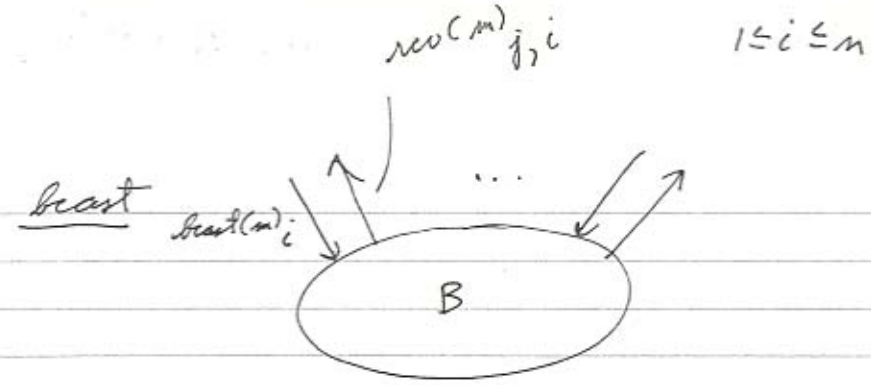
Traces = those seqs β of send + receive actions s.t.

$\exists$ cause function mapping each rec event in β to a preceding send event in β, s.t.

1. same msg — only correct msgs delivered, msg integrity
2. onto — no loss
3. 1-1 — no dup
4. order-preserving — no reo

Other types of channels weaken 2-4, but generally not 1.

Also mcast, bcast, will see later in term, e.g.:



Can receive from self.

Reliable FIFO ~~pp~~ between each pair of processes.

But (in general) different procs can receive in different orders from different processes!

(Formally, write either as trace properties or as fair traces of automaton that keeps separate queues for each i, j pair. bcast puts msg at end of all queues)

mcast:

a variation, $mcast(m)_{(i, I)}$ says to put msg on queues for (i, j), $j \in I$

I — set of procs

bcast, mcast channels often considered with other restrictions: (for all senders)

E.g., consistent total order everywhere

causal order

⋮

— if p_i receives m before bcasting m' then everyone who receives m & m' receives m before m'.

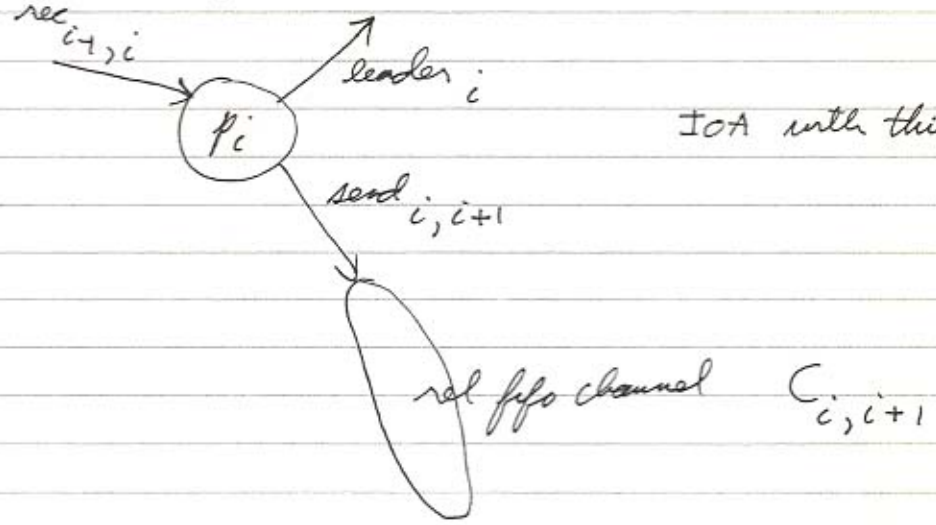
return to this later, under group. comm.

Basic algs in S/R Systems

Leader election:

G = ring, uni or bidirectional

Use local names for nbrs, UIDs



IOA with this interface

Most of the synch algs carry over:

LCR:

Send UID clockwise, throw away results smaller than your own, elect self if UID comes back.

$\boxed{\text{code p. 477}}$ Expresses the asynchrony

Go over

Tasks, fairness

Still works (obviously)

~~Proofs we did before carry over, more or less, but with new computations due to asynchrony:~~

~~Need to consider every step~~

~~Proofs: Use invariants, [illegible] as before.~~

~~But now~~

Proof: ~~[illegible]~~ More work than for synchronous case, though uses many of the same ideas.

Two properties to prove, { one safety: no process $\neq i_{max}$ ever performs leader
{ one liveness: i_{max} eventually performs leader

Safety pf: No process $\neq i_{max}$ ever performs leader.
Recall synch pf based on showing invariant of global state (after any number of rds):

• i_{max} } can't have u_i here (can't get past i_{max})

• i

Previous
Assertion: If $i \neq i_{max}$ + $j \in [i_{max}, i)$ then u_i does not appear in $send_j$. ~~[illegible]~~

~~Now we~~

We use ~~[illegible]~~ essentially the same inv't in the asynch version.
But it has to change a bit:
Now we have finer granularity actions – individual sends + receives.
~~So [illegible]~~
Invariants on global state need to use states of channels too. (queues)

Now:
If $i \neq i_{max}$ + $j \in [i_{max}, i)$ then u_i does not appear in $send_j$

or queue $_{j,j+1}$.

Now can prove as usual, by induction on # of steps in execution.

~~Break into cases by~~

Now the steps are times.

Break into cases based on type of action.

Key case is ~~rcv (v)~~ $_{i-1, i}$, ~~where~~

$rcv(v)_{i_{max}-1, i_{max}}$, must argue that if $v \neq u_{max}$ then v gets discarded.

easy

Liveness: i_{max} eventually outputs leader

Different proof from synchronous case.

There, had just saying exactly where max was after r rounds.

Now no rounds.

More uncertainty, harder to make such definite statements about behavior.

Can establish milestones: here, show inductively on r, $0 \le r \le n-1$, that eventually u_{max} appears in ~~send i_{max}~~

$send_{i_{max}+r}$

Use fairness properties of process & channel automata to prove inductive step.

Complexity: Msgs as before, $O(n^2)$

Time: $O(n(l+d))$

u.b. on local step time for each process

u.b. on time to deliver msg at head of channel