Last time:
Introduced basic automaton model for reactive system components.
I/O automata.
Went over basic def, examples based on simple non-FT concurrent procs
states, start, sig, tros, tasks
+ defined what it means to compose I/O automata
Now finish up with: important properties composition satisfies
- fairness
- proof methods, e.g. invariants
- weak relus
Then go on to cyclic network algos.
Basic composition results (as defined yet)
To show the model works right for describing system in pieces.

Projection + pasting

Thm 1: (Projection)
1. If \( x \in \text{traces}(A) \) then \( \exists ! A_i \in \text{traces}(A_i) \) for every \( i \)
   - delete actions (and following states) for non-actions of \( A_i \)
   - project on \( A_i \)'s state.
2. If \( \beta \in \text{traces}(A) \) then \( \beta \| A_i \in \text{traces}(A_i) \text{ for every } i \in I \)
   - restrict to actions of \( A_i \).

Thm 2: (Pasting)
1. If \( x_i \in \text{traces}(A_i) \) for all \( i \)
   - \( \beta \) is a seq of actions in \text{exit}(A) s.t. \( \beta \| A_i = \text{trace}(x_i) \) for all \( i \)
   - then \( \exists x \in \text{exit}(A) \) s.t. \( \beta = \text{trace}(x) \)
     \[ x \in A_i \] for all \( i \in I \).
2. If \( \beta \) is a seq. of actions in \text{exit}(A) \text{ s.t. } \beta \| A_i \in \text{traces}(A_i) \text{ for all } i \)
   then \( \beta \in \text{traces}(A) \)

Proof: LTPR. Allows compositional reasoning:

Thm 3: (Substitutivity)

Suppose \( A \times A' \) have same exit signature

\[ B = B' \]

Then \( \text{traces}(A \times B') \subseteq \text{traces}(A \times B) \) (a kind of 'implementation relationship')

And suppose \( \text{traces}(A') \subseteq \text{traces}(A) \)

\[ \text{traces}(B') \subseteq \text{traces}(B) \]
Pf: Project, then paste.

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

Left for HW.

Holding: hide \((A)\) = same as \(A\) but with \(E\) reclassified as internal outputs.

Renaming of actions

8.3 Barriers

To complete the basic model thus, need notion of fair exec for an \(TA\), want to say that each task \((\# thread)\) 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 \(x\) defined to be fair if for all \(C \in \text{Tasks}(A)\), one of the following holds:

1. \(x\) is finite + no action of \(C\) is enabled in final state of \(x\)
2. \(x\) is infinite + contains \(\infty\) many steps with acts in \(C\)
3. \(\vdash\) "stages occurrences in which \(C\)
   not enabled"

Fair exec of \(A\)

Fair Trace of \(A\) = Trace of fair exec of \(A\).
Ex: Channel
first example was fair - finite & nothing enabled at end
2nd example was fair - leaves something enabled at end
send send send... infinite & not fair - doesn't give turns to the task.

Ex: Process
was separate tasks for sending to each other process (output)
means must keep trying to deliver, their priority
taxonomy also behaves nicely w/ T composition - results analogous to the unfair case.

Ex: Discuss fair cases of composition of proc & channels
all msgs sent are delivered
after init, send valid values (if changes send new one)
if get whole vector, output correctly

8.4 Properties & Proof Methods
To model inputs & outputs for problems we generally use input & output actions, instead of state vars (as in the systolic model)
Properties to prove:

Invariants: Properties of states, true in all reachable states
Proof by induction (sometimes in batches)
Step granularity is finer than rounds, so proofs get harder
One automaton $A$ "implements" another automaton $B$, in the sense that traces $(A) \subseteq \text{traces} (B)$.

Trace properties: Any property of external schema sequences of an automaton.

Formally, $P = \{\text{signature, set of sequences of acts in the seq.}\}$

\[ \text{in vant only}\]

\[ \text{properly}\]

By $A$ satisfies $P$ if it has the same external signature +

\[ \{ \text{traces} (A) \subseteq \text{traces} (P) \}

\[\text{or}\]

\[\text{fair traces} (A) \subseteq \text{traces} (P)\]

2 possible meanings depending on whether we're interested in liveness or not.

All the problems we'll deal with in async systems can be formulated as trace properties.

We'll usually be concerned about fairness, so we'll use the second notion.

Incidentally, 2 types of trace properties are important enough to have special names:

Safety properties: traces $(P)$ nonempty, prefix closed, limit closed.

Can interpret as "something bad" never happens.
Examples (we'll see)

Consensus: agreement, validity
out of seqs of init or decide actions in which we never get disagreement, never get violation of validity.

Graph searching: correct shortest path.

Mutual exclusion: no 2 simultaneous grants

Why there are safety properties? X doesn't violate

If trace OK then so are all prefixes.
If all prefixes OK, trace OK.

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

Liveness property P:
Every finite seq over \( \Sigma \) (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 \((s, T)\) is a trace property, then
\[
\neg \exists s \ (s, T) \text{ safety property}
\]
\[
\exists s \ (s, L) \text{ liveness property}
\]
So \( T = S \lor L \).

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.
Hierarchical proof

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

Formulate in a series of levels,

- all automate
- abstract spec
- hi-level algo descr.
- more detailed algo descr.

... may continue...

can have high level centralized, lower levels distributed

- insufficient but simple
- optimized
- with large granularity
- fine 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 algo.

Method similar to what I showed for synchronizers algo:

- optimized algo run side by side with unoptimized version
- invariant proved relating the 2 states

Invariant relating 2 states called simulation relation.

Shown using induction.

Want to take advantage of some powerful method for asynch systems.

But there are a few problems...

- Much more freedom allowed in asynch model, in order of 3 steps + in new states
- Harder to determine which executions to compare.
Turns out it's enough to obtain a 1-way relationship between execs, showing for each exec. of the low-level alg., there exists a corresponding exec. of higher-level alg.

**Def:** Assume $A + B$ have same exec. sig. if a binary relation over states $(A)$ and states $(B)$

$$
\leq \text{states}(A) \times \text{states}(B)
$$

(Write $(s, u) \in f$ or $u \in f(s)$.)

Then $f$ is sim reln from $A$ to $B$ provided that

1. If $s \in \text{start}(A)$ then $f(s) \cap \text{start}(B) \neq \emptyset$

   $$(\exists u \in \text{start}(B) \cap f(s))$$

2. If $s, u$ are reachable states of $A$ and $B$ rep., $u \in f(s)$, $$(s, t, s') \in \text{trans}(A)$$ then $\exists$ exec pag $x$ of $B$

starting with $u$, and ending with some $u' \in f(s')$, with

$$\text{trace}(x) = \text{trace}(f)$$

**Theorem:** If there is a sim reln from $A$ to $B$, then traces $(A) \leq \text{traces}(B)$

**If:** For an exec. of $A$, create a corresponding exec. of $B$ using an iterative construction:

- $T_0$, $T_1$, $T_2$, $T_3$, $T_4$
- $S_0$, $S_1$, $S_2$, $S_3$, $S_4$

**If:** For an exec. of $A$, create a corresponding exec. of $B$ using an iterative construction:

- $S_0$, $S_1$, $S_2$, $S_3$, $S_4$
Ex: 2 channels implement one

```
send(m)  \rightarrow  C  \rightarrow  \text{rc}(m)
B  \rightarrow  A  \rightarrow  \text{rc}(m)
```

renaming some actions to "pass"

Claim: $A \times B$ implements $C$, in sense of $\text{Traces}(A) \subseteq \text{Traces}(B)$.

Sim.
- $(s, u) \in \Gamma$ if $u$. queue is correct
  - if $s$, A. queue = $s$, B. queue

If it is the same reln., proves implementation.

How to prove: Check the 2 corresp:

- Start: Empty queues correspond \( \checkmark \)
- Stop: Describe how step corresp.

Helps to give action corresp. first:

- If $p = \text{send}(m)$, use $\text{send}(m)$ in $C$
  (unique state generated from $u$ in $C$)
- If $p = \text{rc}(m)$, use $\text{rc}(m)$ in $C$
- If $p = \text{pass}(m)$, use $\lambda$

Clearly, traces preserved.

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

Break into cases.
\( \Pi = \text{send } (m) \)

No enabling issues (input).

Must check \((s', u') \) of new states.

Have \((s, u) \in \Pi \), add some \(m\) to end of \(u\). queue + \(s\). queue, so still correspond.

\( \Pi = \text{rec } (m) \)

Check \( \text{rec } (m) \) for some \(m\) also enabled in \( \Pi \).

Know \(m\) is first on \(s\).A. queue

By \((s, u) \in \Pi \), \(m\) is first on \(u\). queue.

So enabled in \( \Pi \).

Steps: Effects: Removing heads of both queues correspondance.

\( \Pi = \text{pass } (m) \)

No enabling issues (no bi-level actions)

Just need to show state correspondance

\((s', u') \in \Pi \)

\(s\)

\(s'\)

Know \((s, u) \in \Pi \), 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.
Consider new channel $D$ with duplicate internal action that allows any msg to be duplicated consecutively in the queue.

$$x \leftarrow m \downarrow y \quad \text{duplicates} \quad x \leftarrow m \downarrow m \leftarrow y$$

A single msg \quad two copies

But also modify by adding alternating tag, 0 and 1, to msgs (for duplicating those along with msgs).

Also, remember last msg delivered + throw away following msgs with same tag a msg that is first in the queue but has same tag as the one possibly most recently delivered.

LTTR to formalize.

Key idea of Alternating bit protocol

Claim this channel implements ordinary reliable FIFO channel $A$.

Can show using semi-relax.

5. A queue = collapsed version of s. D queue

\(\text{combine those with same tag into one}\)
\(\text{ignore first msg if same tag as last delivered}\)
\(\text{then erase Tags}\)

Proof: A LTTR

send, $nx \rightarrow$ themselves

dup, throw away, $\rightarrow x$
Asynchronous Networks

Processes communicating via channels

\[ \text{can be point-to-point} \]
\[ \text{cast} \]
\[ \text{broadcast} \]

A sketch of the model, then move to typical problems like leader election, setup search structures (BFS, etc.)

\[ \text{MST} \]

Compare results with such networks

Send receive systems

Use pt-pt channels

A digraph \( G = (V, E) \), process automata associated with nodes

channel automata " " directed edges

No rounds. Allow asynchrony in steps of all components.

Model both proc + channels as 

Process:

\[ \text{send}(m) \]
\[ \text{recv}(m) \]

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

Modeling failures:

\[ \text{stop}_i \]

input, disables all I/O actions

line in external interface

allows problems to be stated in terms of occurrences of failures.

(generally, liveness properties)
Channels:

Send (write) \( \rightarrow \) \( \text{send}(n) \) \( i \), \( j \)

Receive \( \rightarrow \) \( \text{recv}(n) \) \( i \), \( j \)

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

- reliable FIFO
- various kinds of less reliable:
  allowing losses, duplicates, reading, e.g.

This is just the channel already described - state is queue,
Read adds, new removes head

Processes: all receives in 1 task.

Alternative description of channel: Interface + trace constraint

Trace property:

Traces = those seqs \( \beta \) of send & receive actions s.t.

- \( \exists \) base function mapping each new event in \( \beta \) to a preceding
  send event in \( \beta \), s.t.:

1. Base ning only cannot send delivered
2. onto no loss
3. 1-1 no dup
4. order-preserving no see

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

Also meant, best will see data in turn, e.g.
Reliable FIFO flow between each pair of processes.

But (in general) different pairs can receive in different orders from different processes.

Essentially, write either as trace properties or as fair traces of automata that keeps separate queues for each $i,j$ pair.

Each put goes at end of all queues.

A variation: `meast (m)` $(i, I)$ says to put just on queues for $(i,j)$, $j \in I$

set of pros

Beast, meast channels often considered with other restrictions:
E.g., consistent total order everywhere (for all senders)
causal order

if $p_i$ receives $m$ before $m$ then $p_i$ receives $m$ after every $m$ received
$m$ before $m$

return to this latter under group comm.
Basic Alg in S/R Systems

Leader election:
- C = ring, uni or bidirectional
- Use local names for subs, UIDs

Most of the sync algos carry over:

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

Expands the anonymity
- to over

Tasks, fairness

Still works (obviously)

Proofs we did before carry over, same or less, but with new complications due to anonymity.

Need to consider very first

Proof: Use invariants with strictly less than before.
Prof: More work than for synchronous case, though uses many of the same ideas.

Two properties to prove: one safety, one liveness. 

Safety pf: No process + \( i_{\text{max}} \) ever prefers leader.

Recall loop pf based on sharing invariant of global state (after any number of reqs):

\[
\begin{align*}
\text{for } i, j \in \overline{i_{\text{max}} : i} \text{ } i_{\text{max}} \text{ can't be } & \text{ here (can't get part \( i_{\text{max}} \))} \\
\text{Previos} & \text{ Assertion: if } i + j \in \overline{i_{\text{max}} : j} \text{ then } u_j \text{ does not appear in } \text{send}_i.
\end{align*}
\]

From now on:

We use essentially the same loop in the asynchronous version.

But it has to change a bit:

Now we have finer granularity action - individual sends & receives.

Invariants on global state need to use states of channels too. (queues)

New:

\[
\text{if } i + j \in \overline{i_{\text{max}} : j} \text{ then } u_j \text{ does not appear in } \text{send}_i \text{ on queue } j_{\text{f+1}}.
\]
Now can prove as usual, by induction on # of steps in execution.

Break into cases.

Now the steps are taken.

Break into cases based on type of action.

Key case is \(0 < n < 1\).

\[ \text{new}(u) \]

\[ i = \min \{ i \mid \text{new}(u) \} \]

\[ \text{max}(v) \]

must argue that if \( v \neq \max \)

then \( v \) gets discarded.

License: \( \max \) eventually outputs leader.

Different proof from synchronous case.

Thus, had meant saying exactly where \( \max \) was after \( n \) rounds.

Now no rounds.

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

It can establish probabilities; here, show individually on one channel that eventually \( \max \) appears in head:

send:

\[ \text{max} + 1 \]

use fairness properties of process \( v \) channel automata to prove inductive step.

Completeness: \( M \) as before, \( O(n^2) \)

Time: \( O(n(n+k)) \)

u.d. on tie to deliver msg at head

for each process of channel.