Last time:
introduced basic automation model for reactive system components.
- I/O automata.

Went over basic defs, examples based on simple non-FT consensus procs
- states, start, sig, tras, 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 asyncio networking algo.
Basic composition results (as discussed yet)

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

Projection + pasting:

\[ \text{Shm 1: (Projection)} \]

1. \( \text{If } \alpha \in \text{exces}(A) \text{ then } \alpha \cdot \Pi_i \in \text{exces}(A_i) \text{ for every } i \)

   (delete actions (4 following states) for non-actions of \( A_i \)) + project on \( A_i \)’s state.

2. \( \text{If } \beta \in \text{traces}(A) \text{ then } \beta \cdot \Pi_i \in \text{traces}(A_i) \text{ for every } i \in I \)

   (*note: to actions of \( A_i \))

\[ \text{Shm 2: (Pasting)} \]

1. \( \text{If } \lambda_i \in \text{exces}(A_i) \text{ for all } i \),

   \( \beta \) is a seq of actions in \( \text{ext}(A) \) s.t. \( \beta \cdot \Pi_i = \text{trace}(\lambda_i) \) for all \( i \)

   \( \exists \) \( \lambda \) of \( A \) s.t. \( \beta = \text{trace}(\lambda) \) + \( \lambda_i = \lambda \cdot \Pi_i \) for all \( i \in I \).

2. \( \text{If } \beta \) is a seq of actions in \( \text{ext}(A) \) + \( \beta \cdot \Pi_i \in \text{traces}(A_i) \text{ for all } i \)

   \( \beta \in \text{traces}(A) \)

Proofs LTRR. Allows compositional reasoning:

\[ \text{Shm 3: (Substitutivity)} \]

Suppose \( A \) \& \( A' \) have same ext. signature

\[ \text{Then } \text{traces}(A \times B') \subseteq \text{traces}(A \times B) \]

\( \text{and } \text{traces}(A') \subseteq \text{traces}(A) \)

\( \text{traces}(B') \subseteq \text{traces}(B) \)

(a kind of "implementation relationship")
Pf: project, then paste.

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

Left for HW.

Rerouting: hide \( \varepsilon \) = same as \( \varepsilon \), but with \( \varepsilon \) reclassified as internal

8.3 Fairness

To complete the basic model description, need notion of fair exec for an FTA.

Want to say that one task (or thread) keeps getting terms 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 \( \alpha \) defined to be fair if for all \( C \in \text{Tasks}(A) \), one of the following holds:

1. \( \alpha \) is finite and no action of \( C \) is enabled in final state of \( A \)
2. \( \alpha \) is infinite and contains as many steps with acts in \( C \) as states occurrences in which \( C \) not enabled.

Fair exec of \( A \)

Fair trace of \( A \) = trace of fair exec of \( A \).
Ex: Channel

1st example case 1 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 identify these factors.

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

Ex: Denies fair access of composition of processes and channels.

All messages are delivered.

After init, send real future (if changes send new one)

If get whole sector, output future.

8.4. Properties & Proof Methods

To model inputs & outputs for problems we generally use input & output actions, instead of state variables (as in the sync 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.
One automation A "implements" another automation B, in the sense that traces (A) \subseteq traces (B).

Trace properties: Any property of external behavior sequences traces of an automation.

\[ \text{Property} \]

Say A satisfies P if it has the same external signature +

\[ \begin{align*}
\text{traces} (A) & \subseteq \text{traces} (P) \\
\text{fairtraces} (A) & \subseteq \text{traces} (P)
\end{align*} \]

\[ \text{2 possible meanings} \]

Depending on whether we're interested in fairness or not.

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

\[ \text{2 possible meanings} \]

& 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
- 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 or more simultaneous grants

To see why there are safety properties: S 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.

Safety 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 if the feature.

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

Formally, if \( (\text{sig}, T) \) is a trace property, then

\[
I \ (\text{sig}, S) \text{ safety property}
\]

*\( I \ (\text{sig}, L) \) liveness property
*\( T = S \cup L \).

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 proofs

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

Formulate in a series of levels,
all automata

abstract spec

hi-level algo description

more detailed algo descr.

may continue.

Can have high level centralized; lower levels distributed

inefficient 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 algs.

Method similar to what I showed for synchronizers algs:

optimized alg 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 async systems.

But there are a few problems...

Much more freedom allowed in async model, in order of

steps & new states

Hardest to determine which executions to compare.
Turns out it's enough to obtain a 1-1 mapping 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 ext. maj.

Let a binary relation over states(A) and states(B)

\[ \subseteq \text{states}(A) \times \text{states}(B) \]

(Write \((s, \mu) \in f \) or \( \mu \in f(s) \))

Then \( f \) is sim reln from A to B provided that

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

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

2. If \( s, \mu \) are reachable states of A and B resp., \( \mu \in f(s) \)

   \( s, (\mu, \tau, s') \in \text{trans}(A) \) then \( \exists \text{ exec. prog. } x \) of B

   starting with \( \mu \), and ending with \( \text{ some } \mu' \in f(s') \),

   with Trace(x) = Trace(f)

   Explain:

   \[ \begin{array}{c}
   s \quad \mu \quad \ldots \ldots \quad \mu' \\
   \uparrow \\
   f \downarrow \\
   \sigma \quad \tau \\
   \end{array} \]

   can be 0, 1 or more steps

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

 proofs:

1. \( \forall s \in \text{exec. of } A \) create a corresponding exec. of B using an iterative construction:

   \[ \begin{array}{c}
   s_0 \\
   \vdots \\
   s_k \\
   \vdots \\
   s_{\infty} \\
   \end{array} \]

   Gives full correct exec.

   Easy to see correct.

   Can be finite or infinite.
Ex: 2 channels implement one

send(m)  \rightarrow  C  \rightarrow  rec(m)

send(m)  \rightarrow  B  \rightarrow  pass(m)  \rightarrow  A  \rightarrow  rec(m)

\uparrow

renaming some actions to "pass"

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

Syn reln: (s, u) \in I \iff u, queue is correct

\iff s, A, queue + s, B, queue

If it is not a pair, reln, processes implementation.

How to parse: Check the 2 cases:

Start: Empty queues correspond ✓

Step: Describe step corresp.

Helps to give action corresp. first:

If \( \Pi = \text{send}(m) \), use \text{send}(m) in C

(unique state generated from \( u, m \) in C)

If \( \Pi = \text{rec}(m) \), use \text{rec}(m) in C

If \( \Pi = \text{pass}(m) \), use \lambda

Clearly traces preserved.

Show correspondence:

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

Breaks into cases:
\( \tau^r = \text{send } (m) \)

- No enabling issues (input).
- Must check \((s', u') \) if new states.
- Have \((s, u) \in \mathcal{E}_f\) add same \(m\) to end of \(u\) queue + \(s\) B. queue
  so still correspond.

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

- Check \( \text{rec } (m) \) for same \(m\) also enabled in \(u\).
- Know \(m\) is first on \(S.A.\) queue
- By correspondence \((s, u) \in \mathcal{E}_f\), \(m\) is first on \(u\) queue.
  So enabled in \(u\).

**Step 2: Effects**: Removing heads of both queues correspondence.

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

- No enabling issues (no hi-level actions)
- Just need to show state correspondence \((s', u') \in \mathcal{E}_f\)

Know \((s, u) \in \mathcal{E}_f\), 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.
Consider new channel \( D \) with duplicate function action that allows any msg to be duplicated consecutively in the queue.

\[
\begin{array}{c}
\times m y \\
\downarrow \\
\times m m y \\
\uparrow \\
a single msg \\
two copies
\end{array}
\]

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

Also, remember last tag delivered \( D \) 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 finalize.

Key idea of alternating bit instead.

Claim this channel implements ordinary reliable FIFO channel A,

trace inclusion.

Can show using sim. reln.

25. A. queue = collapsed version of \( D \), queue

combine those with same tag into one

(ignores first msg if same tag as last delivered)

then erase tags

Proof: A LTTR.

send, rec \rightarrow\text{ themselves}

dup, throwaway \rightarrow x
Asynchronous Networks

Processes communicating via channels can be point-to-point or multicast, best

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

Compare results with such networks.

Send/receive system using pt-pt channels.

Directed graph $G = (V, E)$, process automata associated with nodes

Channel automata for directed edges with rounds. Allow asynchrony in steps of all components.

Model both process and channels as $\zeta_0|\zeta_5$

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

Modeling failures:

- e.g. $\text{stop } i \rightarrow g_i$, input, disable all I/O actions, reset since in external interface,

allow problems to be stated in terms of occurrences of failures (generally, liveness properties)
Channels:
- Send (receive): $send(m) \rightarrow C_{ij}$, $rec(m) \rightarrow C_{ij}$

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.
  - (will come back to this later)
  - for now, just consider rel. FIFO

This is just the channel already described - state is queue:
- Send adds, new received leads
- Listeners: all receives in 1 task

Alternate description of channel:
- Interface + trace constraints
  - Trace property.

Traces: those seqs $\beta$ of send + receive actions s.t.
- $f$ cause function mapping each new event in $\beta$ to a preceding send event in $\beta$, s.t.
  1. Some msg only carried msg delivered
  2. onto msg integrity
     - no loss
  3. 1-1
     - no dup
  4. Order-preserving
     - no rec

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

Also meant, boost, will see later in turn, e.g.: 
\textbf{beast} \texttt{read(m) i} \quad 1 \leq i \leq n

\begin{itemize}
  \item \texttt{B}
  \item Can receive from self
\end{itemize}

Reliable FIFO app between each pair of processes.
But (in general) different processes can receive in different orders from different processes.

(Formerly, wrote either as trace properties or as fair traces of automaton that keeps separate queues for each \( i,j \) pair. Beast puts msg at end of all queues.

\texttt{meast}:

A variation, \texttt{meast(m)} \((i, I)\) says to put msg on queues for \((i,j) \quad j \in I\)

\texttt{beast, meast} channels often considered with other restrictions:

\begin{itemize}
  \item \texttt{causal order}:
    \begin{itemize}
      \item if \( p \) receives \( m \) before \( b \), then \( b \) receives \( m \) after \( p \).
    \end{itemize}
\end{itemize}

\texttt{return to this later under group comm.}
Basic algo in S/R systems

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

[Diagram showing leader election process with arrows indicating communication]

IOA with this interface

Most of the synch algo carry over:

LCR:
- Send VID clockwise, throw away results smaller than your own,
- elect self if VID comes back!

[Note: code p. 477] Expresses the ambiguity

Tasks, fairness

Still needs (obviously)

... thugs we list before any other, make a list, but with new complexities due to ambiguity.

Need to consider very fast

Pros: Use incumbent, needs bandwidth as before.

But...
Proof: 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 \( i \) ever performs leader.

Recall synch Pf based on showing invariant of global state (after any number of rds):

\[ \{ i \text{ max} \} \text{ can't have } \forall j \text{ here (can't get part } i \text{ max)} \]

Previous Assertion: If \( i = i \text{ max} + j \in [i \text{ max}, i) \) then \( \nu \) does not appear in send \( j \).

Now:

- Essentially the same just in the asynch version.
- But it has to change a bit:
- Now we have finer granularity: action - individual sends & receives.
- Invariances on global state need to use states of channels too. (queues)
- New: If \( i = i \text{ max} + j \in [i \text{ max}, i) \) then \( \nu \) does not appear in send \( j \) or queue \( j \), \( j+1 \).
How 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

rev(v) \leq \frac{i_{\text{max}} - 1}{i_{\text{max}}} \quad \text{must argue that if } v \neq \text{max} \quad \text{then } v\text{ gets discarded.}

Remarks: \( i_{\text{max}} \) eventually outputs leader

Different proof from synchronous case.

There's bad limit saying exactly when \( \text{max} \) was after \( n \) rounds.

Now no rounds.

Max uncertainty, leader to make such definite statements about behavior.

It can establish milestones: how, show inductively on \( i \), obtaining that eventually \( \text{max} \) appears in head.

read:

\[ \text{max} + n \]

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

Complexity: \( M = n \text{ as before}, \ O(n^2) \)

Size: \( O(n (1+d)) \)

U.B. on local step tie

For each process of channel