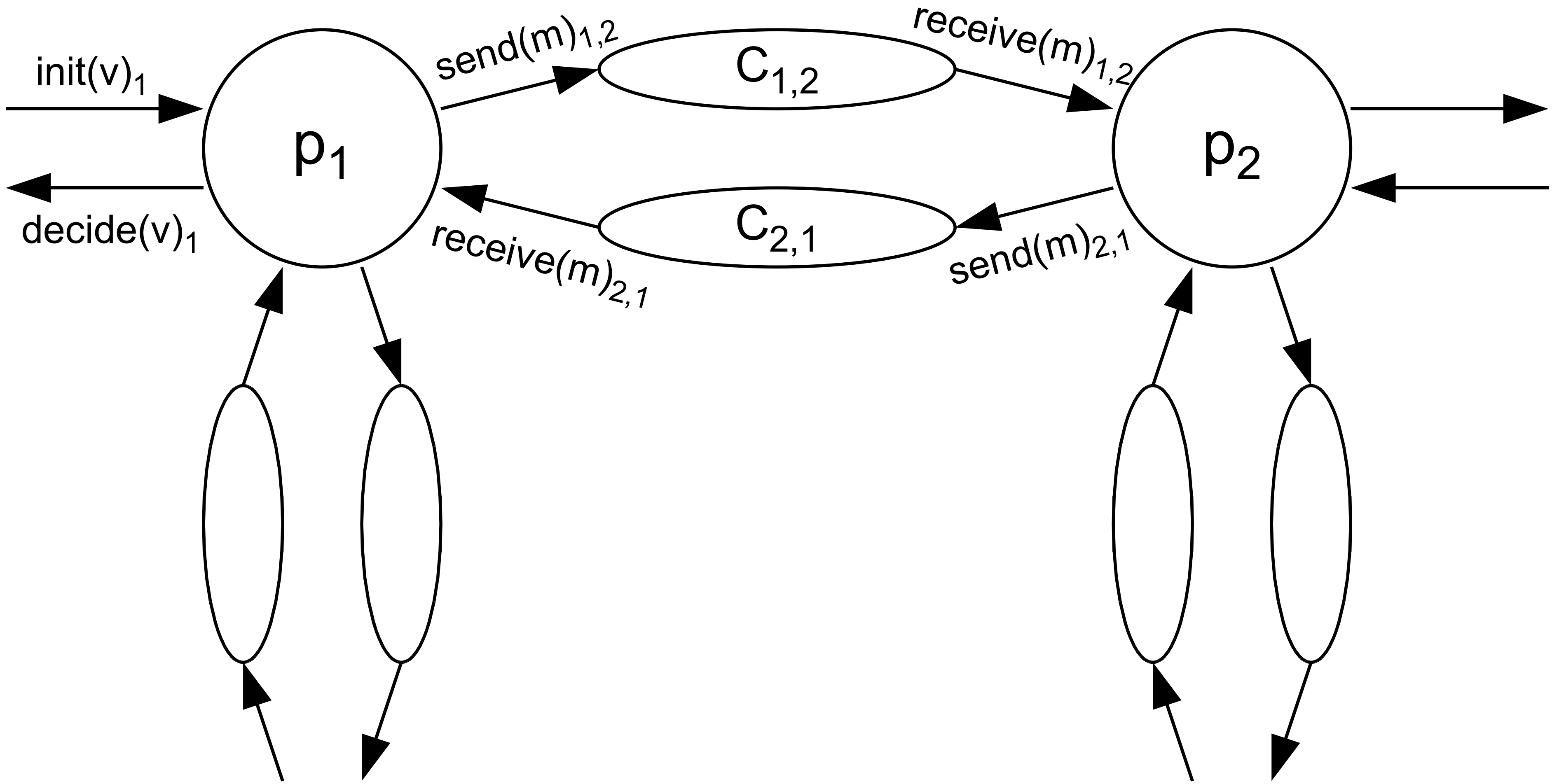
6.852 Lecture 7

- Asynchronous systems
- Formal model
  - I/O automata
  - behaviors
  - simulations
  - composition
- Reading: Chapter 8

Asynchronous systems

- No timing assumptions
  - no rounds
- Asynchronous networks
  - nodes communicating via channels
- Asynchronous shared memory
  - processes communicating via shared objects

Asynchronous network



Specifying problems and systems

- Processes and channels are automata
  - take **actions** to change state
  - reactive
    - interact with environment via input and output actions
    - not just map from input values to output values
- Behavior
  - we observe **externally visible** actions
    - state is hidden
  - interleaving semantics
    - behavior is sequence of actions
  - problems specify allowable behaviors

Input/output automaton

- General mathematical model
  - very little structure
- Designed for “structured” system description
  - composition
  - hierarchical description/reasoning
- Supports good proof techniques
  - invariants
  - simulation relations
  - compositional reasoning

Input/output automaton

- State transition system
  - transitions labeled by actions
- Actions classified as input, output, internal
  - input, output are **externally visible**
  - output, internal are **locally controlled**

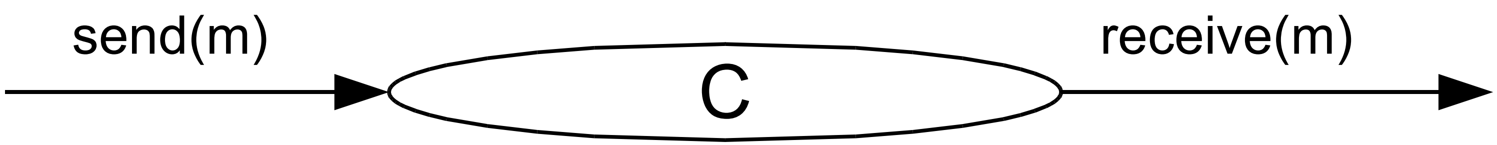
Input/output automaton

- sig(A) = (in(A), out(A), int(A))
  - input, output, internal actions (disjoint)
  - acts(A) = in(A) $\cup$ out(A) $\cup$ int(A)
- states(A)
- start(A) $\subseteq$ states(A)
- trans(A) $\subseteq$ states(A) $\times$ acts(A) $\times$ states(A)
  - input-enabled
- tasks(A), partition of local(A)
  - needed for liveness

Input/output automaton

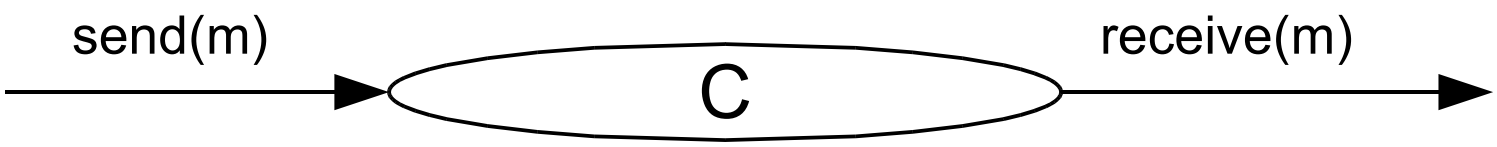
- A **step** of an automaton is an element of trans
- Action π is **enabled** in a state s
  - if there is a step (s, π,s') for some s'
- I/O automata must be **input-enabled**
  - every input action is enabled in every state
  - captures idea that automaton cannot control inputs
  - enables compositional reasoning
- tasks correspond to “threads of control”
  - used to define fairness
  - needed to guarantee liveness

Channel automaton



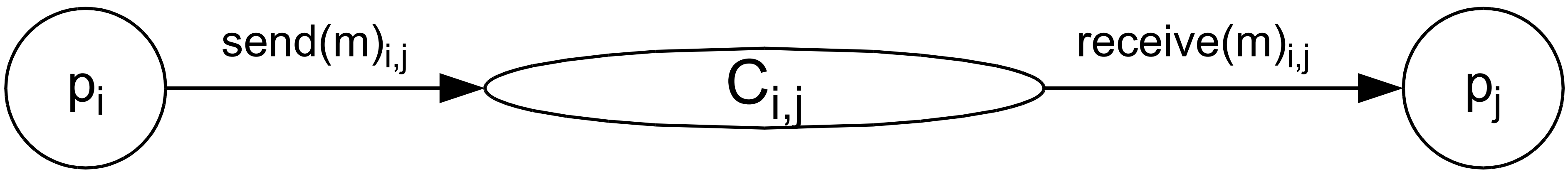
- Reliable unidirectional FIFO channel for 2 processes
  - fix message “alphabet” M
- signature
  - input actions: send(m) for $m \in M$
  - output actions: receive(m) for $m \in M$
  - no internal actions
- states
  - **queue**: FIFO queue of M, initially empty

Channel automaton



- trans
  - send(m)
    - effect: add m to (end of) **queue**
  - receive(m)
    - precondition: m is at head of **queue**
    - effect: remove head of **queue**
- tasks
  - all receive actions in one task

Channel automaton



- trans
  - $send(m)_{i,j}$
    - effect: add m to (end of) **queue**
  - $receive(m)_{i,j}$
    - precondition: m is at head of **queue**
    - effect: remove head of **queue**
- tasks
  - all receive actions in one task

Executions

- An I/O automaton executes as follows:
  - start at some start state
  - repeatedly take step from current state to new state
- Formally, an **execution** is a sequence:
  - $s_0\ \pi_1\ s_1\ \pi_2\ s_2\ \pi_3\ s_3\ \pi_4\ s_4\ \pi_5\ s_5$ ... (if finite, end in state)
  - s_0 is a start state
  - $(s_i, \pi_{i+1}, s_{i+1})$ is a step (i.e., in trans)

λ, send(a), a, send(b), ab, receive(a), b, receive(b), λ

Executions

- An I/O automaton executes as follows:
  - start at some start state
  - repeatedly take step from current state to new state
- Formally, an ~~**execution**~~ **execution fragment** is a sequence:
  - $s_0\ \pi_1\ s_1\ \pi_2\ s_2\ \pi_3\ s_3\ \pi_4\ s_4\ \pi_5\ s_5$ ... (if finite, end in state)
  - ~~s_0 is a start state~~
  - $(s_i, \pi_{i+1}, s_{i+1})$ is a step (i.e., in trans)

λ, send(a), a, send(b), ab, receive(a), b, receive(b), λ

Invariants and reachable states

- A state is **reachable** if it appears in some execution.
  - equivalently, at the end of some finite execution
- An **invariant** is a predicate that is true on every reachable state.
  - main tool for proving properties of concurrent algorithms
  - typically prove by induction on length of execution

Traces

- A **trace** of an execution is the subsequence of external actions in the execution
  - denoted trace(α), where α is an execution
  - models "observable behavior"

λ, send(a), a, send(b), ab, receive(a), b, receive(b), λ

send(a), send(b), receive(a), receive(b)

Trace properties

- A **trace property** P is a pair of:
  - sig(P): external signature (i.e., no internal actions)
  - traces(P): set of sequences of actions in sig(P)
  - can specify allowable behaviors
- Automaton A satisfies trace property P if
  - extsig(A) = sig(P) and traces(A) $\subseteq$ traces(P)
  - extsig(A) = sig(P) and fairtraces(A) $\subseteq$ traces(P)

Automata as specifications

- Every I/O automaton specifies a trace property
  - (extsig(A), traces(A))
  - we can use an automaton as a problem specification
- Hierarchical proofs
  - important strategy for proving correctness of complex asychronous distributed algorithms
  - automaton A **implements** B if
    - extsig(A) = extsig(B)
    - traces(A) $\subseteq$ traces(B)
  - define a series of automata, each implementing the next
    - first automaton models algorithm/system; last captures spec

Simulation relations

- Most common method to prove one automaton implements another
- Similar to technique for synchronous algorithms
  - map states in one to states of other
  - show correspondence holds initially, is preserved each round
  - also similar to abstraction function for data type implementation
- R is a **simulation relation** from A to B provided:
  - $s_A \in$ start(A) implies there exists $s_B \in$ start(B) such that s_A R s_B
  - if s_A, s_B are reachable states of A and B, s_A R s_B and (s_A, π, s'_A) is a step, then there exists an exec fragment β starting with s_B and ending in s'_B such that s'_B R s'_A and trace(π) = trace(β)

Simulation relations

$$
\begin{tikzcd}
s_B \arrow[r, dashed, "\beta"] \arrow[d, dash, "R"] & s'_B \arrow[d, dash, dashed, "R"] \\
s_A \arrow[r, "\pi"] & s'_A
\end{tikzcd}
$$

- R is a **simulation relation** from A to B provided:
  - $s_A \in start(A)$ implies there exists $s_B \in start(B)$ such that $s_A \; R \; s_B$
  - if s_A, s_B are reachable states of A and B, $s_A \; R \; s_B$ and (s_A, π, s'_A) is a step, then there exists an exec fragment β starting with s_B and ending in s'_B such that $s'_B \; R \; s'_A$ and $trace(\pi) = trace(\beta)$

Simulation relations

- Theorem: If there is a simulation relation from A to B then traces(A) $\subseteq$ traces(B).

$$s_{0,A} \xrightarrow{\pi_1} s_{1,A} \xrightarrow{\pi_2} s_{2,A} \xrightarrow{\pi_3} s_{3,A} \xrightarrow{\pi_4} s_{4,A} \xrightarrow{\pi_5} s_{5,A}$$

Simulation relations

- Theorem: If there is a simulation relation from A to B then traces(A) $\subseteq$ traces(B).

$$\begin{tikzcd}
s_{0,B} \arrow[d, dash, dashed, "R"] \\
s_{0,A} \arrow[r, "\pi_1"] & s_{1,A} \arrow[r, "\pi_2"] & s_{2,A} \arrow[r, "\pi_3"] & s_{3,A} \arrow[r, "\pi_4"] & s_{4,A} \arrow[r, "\pi_5"] & s_{5,A}
\end{tikzcd}$$

Simulation relations

- Theorem: If there is a simulation relation from A to B then traces(A) $\subseteq$ traces(B).

$$
\begin{tikzcd}
s_{0,B} \arrow[r, dashed, "\beta_1"] \arrow[d, dash, "R"] & s_{1,B} \arrow[d, dash, dashed, "R"] & & & & \\
s_{0,A} \arrow[r, "\pi_1"] & s_{1,A} \arrow[r, "\pi_2"] & s_{2,A} \arrow[r, "\pi_3"] & s_{3,A} \arrow[r, "\pi_4"] & s_{4,A} \arrow[r, "\pi_5"] & s_{5,A}
\end{tikzcd}
$$

Simulation relations

- Theorem: If there is a simulation relation from A to B then traces(A) $\subseteq$ traces(B).

$$\begin{tikzcd}
s_{0,B} \arrow[r, "\beta_1"] \arrow[d, dash, "R"] & s_{1,B} \arrow[r, dashed, "\beta_2"] \arrow[d, dash, "R"] & s_{2,B} \arrow[r, dashed, "\beta_3"] \arrow[d, dash, dashed, "R"] & s_{3,B} \arrow[r, dashed, "\beta_4"] \arrow[d, dash, dashed, "R"] & s_{4,B} \arrow[r, dashed, "\beta_5"] \arrow[d, dash, dashed, "R"] & s_{5,B} \arrow[d, dash, dashed, "R"] \\
s_{0,A} \arrow[r, "\pi_1"] & s_{1,A} \arrow[r, "\pi_2"] & s_{2,A} \arrow[r, "\pi_3"] & s_{3,A} \arrow[r, "\pi_4"] & s_{4,A} \arrow[r, "\pi_5"] & s_{5,A}
\end{tikzcd}$$

Fairness

- Recall tasks(A): partition of local(A)
  - task corresponds to "thread of control"
  - used to define "fair" executions
    - a "thread" that is continuously enabled gets to take a step
  - needed to prove liveness
- Formally, an execution α is **fair** to $C \in$ tasks (A) if:
  - α is finite and C is not enabled in final state
  - α is infinite and either
    - infinitely many events in C occur in α; or
    - C is not enabled in infinitely many states in α

Fairness

- Example: Channel
  - only one task (all receive actions)
  - an finite execution of Channel is fair iff **queue** is empty
  - Is every infinite execution of Channel fair?
- Recall alternative defn of “A satisfies P”
  - if extsig(A) = sig(P) and fairtraces(A) $\subseteq$ traces(P)
  - weaker than traces(A) $\subseteq$ traces(P)
- Fairness is a **liveness** property

Safety and liveness

- **Safety** property: “bad” thing doesn't happen
  - nonempty
  - prefix-closed
  - limit-closed
- **Liveness** property: “good” thing happens eventually
  - every finite sequence over acts(P) has an extension (is a prefix of) some sequence in traces(P)

Composition

- "Put multiple automata together"
  - output actions of one may be input actions of others
- Look first at composing two automata
  - generalize to composing infinitely many automata (in book)
- Recall:
  - sig(A) = (in(A), out(A), int(A))
  - local(A) = out(A) ∪ int(A)
- Two automata A and B are **compatible** if
  - local(A) and local(B) are disjoint
  - int(A) and acts(B) are disjoint
  - int(B) and acts(A) are disjoint

Composition

- $A \times B$, composition of A and B
  - $int(A \times B) = int(A) \cup int(B)$
  - $out(A \times B) = out(A) \cup out(B)$
  - $in(A \times B) = in(A) \cup in(B) - (out(A) \cup out(B))$
  - $states(A \times B) = states(A) \times states(B)$
  - $start(A \times B) = start(A) \times start(B)$
  - $trans(A \times B)$: includes (s, π, s') iff
    - $(s_A, \pi, s'_A) \in trans(A)$ if $\pi \in acts(A)$; $s_A = s'_A$ otherwise
    - $(s_B, \pi, s'_B) \in trans(B)$ if $\pi \in acts(B)$; $s_B = s'_B$ otherwise
  - $tasks(A \times B) = tasks(A) \cup tasks(B)$

Composition

- Projection
- Execution pasting
- Trace pasting

Next lecture

- Finish up composition
  - theorems
  - examples
- Basic asynchronous network algorithms
  - Chapter 15