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

p₁

init(v)₁

decide(v)₁

send(m)₁,₂

C₁,₂

receive(m)₁,₂

C₂,₁

send(m)₂,₁

receive(m)₂,₁

p₂
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

- \( \text{sig}(A) = (\text{in}(A), \text{out}(A), \text{int}(A)) \)
  - input, output, internal actions (disjoint)
  - \( \text{acts}(A) = \text{in}(A) \cup \text{out}(A) \cup \text{int}(A) \)

- \( \text{states}(A) \)

- \( \text{start}(A) \subseteq \text{states}(A) \)

- \( \text{trans}(A) \subseteq \text{states}(A) \times \text{acts}(A) \times \text{states}(A) \)
  - input-enabled

- \( \text{tasks}(A) \), partition of local(\(A\))
  - needed for liveness
Input/output automaton

- A **step** of an automaton is an element of transitions.
- Action $\pi$ is **enabled** in a state $s$ if there is a step $(s, \pi, s')$ for some $s'$.
- I/O automata must be **input-enabled**:
  - every input action is enabled in every state,
  - captures the idea that the 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: $\text{send}(m)$ for $m \in M$
  - output actions: $\text{receive}(m)$ 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)\textsubscript{i,j}
    - effect: add m to (end of) queue
  - receive(m)\textsubscript{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:
  – \( s_0 \pi_1 s_1 \pi_2 s_2 \pi_3 s_3 \pi_4 s_4 \pi_5 s_5 \ldots \) (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)

\( \lambda, \text{send}(a), a, \text{send}(b), ab, \text{receive}(a), b, \text{receive}(b), \lambda \)
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:
  - $s_0 \pi_1 s_1 \pi_2 s_2 \pi_3 s_3 \pi_4 s_4 \pi_5 s_5 \ldots$ (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)

$\lambda$, send(a), a, send(b), ab, receive(a), b, receive(b), \lambda
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”

\[ \lambda, \text{send}(a), a, \text{send}(b), ab, \text{receive}(a), b, \text{receive}(b), \lambda \]

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) ⊆ traces(P)
  – extsig(A) = sig(P) and fairtraces(A) ⊆ traces(P)
Automata as specifications

- Every I/O automaton specifies a trace property
  - $(\text{extsig}(A), \text{traces}(A))$
  - we can use an automaton as a problem specification

- Hierarchical proofs
  - important strategy for proving correctness of complex asynchronous distributed algorithms
  - automaton $A$ implements $B$ if
    - $\text{extsig}(A) = \text{extsig}(B)$
    - $\text{traces}(A) \subseteq \text{traces}(B)$
  - 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:
  – $s_A \in \text{start}(A)$ implies there exists $s_B \in \text{start}(B)$ such that $s_A \sim s_B$
  – if $s_A, s_B$ are reachable states of A and B, $s_A \sim s_B$ and $(s_A, \pi, s'_A)$ is a step, then there exists an exec fragment $\beta$ starting with $s_B$ and ending in $s'_B$ such that $s'_B \sim s'_A$ and $\text{trace}(\pi) = \text{trace}(\beta)$
Simulation relations

- $R$ is a simulation relation from $A$ to $B$ provided:
  - $s_A \in \text{start}(A)$ implies there exists $s_B \in \text{start}(B)$ 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 $(s_A, \pi, s'_A)$ is a step, then there exists an exec fragment $\beta$ starting with $s_B$ and ending in $s'_B$ such that $s_B \, R \, s'_A$ and $\text{trace}(\pi) = \text{trace}(\beta)$
Simulation relations

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

---

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

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

\[
\begin{align*}
S_{0,A} & \xrightarrow{\pi_1} S_{1,A} \xrightarrow{\pi_2} S_{2,A} \xrightarrow{\pi_3} S_{3,A} \xrightarrow{\pi_4} S_{4,A} \xrightarrow{\pi_5} S_{5,A} \\
S_{0,B} & \xrightarrow{R} S_{1,A} \xrightarrow{\pi_2} S_{2,A} \xrightarrow{\pi_3} S_{3,A} \xrightarrow{\pi_4} S_{4,A} \xrightarrow{\pi_5} S_{5,A}
\end{align*}
\]
Simulation relations

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

---

![Simulation Diagram]

- $s_{0,A} \xrightarrow{\pi_1} s_{1,A} \xrightarrow{\pi_2} s_{2,A} \xrightarrow{\pi_3} s_{3,A} \xrightarrow{\pi_4} s_{4,A} \xrightarrow{\pi_5} s_{5,A}$
- $s_{0,B} \xrightarrow{\beta_1} s_{1,B}$
- $R$
Simulation relations

- Theorem: If there is a simulation relation from A to B then \(\text{traces}(A) \subseteq \text{traces}(B)\).
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 $\alpha$ is **fair** to $C \in \text{tasks (A)}$ if:
  – $\alpha$ is finite and $C$ is not enabled in final state
  – $\alpha$ is infinite and either
    • infinitely many events in $C$ occur in $\alpha$; or
    • $C$ is not enabled in infinitely many states in $\alpha$
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) ⊆ traces(P)
  - weaker than traces(A) ⊆ 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:
  - \( \text{sig}(A) = (\text{in}(A), \text{out}(A), \text{int}(A)) \)
  - \( \text{local}(A) = \text{out}(A) \cup \text{int}(A) \)

• Two automata \( A \) and \( B \) are **compatible** if
  - \( \text{local}(A) \) and \( \text{local}(B) \) are disjoint
  - \( \text{int}(A) \) and \( \text{acts}(B) \) are disjoint
  - \( \text{int}(B) \) and \( \text{acts}(A) \) are disjoint
Composition

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

- Projection
- Execution pasting
- Trace pasting
Next lecture

• Finish up composition
  – theorems
  – examples

• Basic asynchronous network algorithms
  – Chapter 15