Goals:
- Introduce state transition systems and the computational model (UNITY)
- Define weak and strong fairness assumptions for program execution

Reading:
- P. Sivilotti, *Introduction to Distributed Algorithms*, Chapter 2
Example: RoboFlag (D’Andrea, Cornell)

Robot version of “Capture the Flag”

- Teams try to capture flag of opposing team without getting tagged
- Mixed initiative system: two humans controlling up to 6-10 robots
- Limited BW comms + limited sensing
Distributed Decision Making: “RoboFlag Drill”

Task description
- Incoming robots should be blocked by defending robots
- Incoming robots are assigned randomly to whoever is free
- Defending robots must move to block, but cannot run into or cross over others
- Allow robots to communicate with left and right neighbors and switch assignments

Goals
- Would like a provably correct, distributed protocol for solving this problem
- Should (eventually) allow for lost data, incomplete information

Questions
- How do we model a (distributed) protocol?
- Given a protocol, how do we prove specs?
- How do we design the protocol given specs?
Programs

Programs (also called “processes”) consist of

- A set of typed variables, possibly with initial values
- Assignment statements (or “actions”)
  - Fatbar ([]) separates assignments
  - Actions can be executed in any order (nondeterministic)

Visualization of programs as graphs

- Each state (possible value of variables) is a vertex
- (Directed) Edges represent assignments (actions) that change state

“Skip”

- All programs implicitly contain the skip assignment, which leaves the state of the program unchanged

**Program Trivial**

```
var x, y : number
initially x ≠ 2
assign
    x := 2
[] y := f(7)
```

**Initial state**
Actions

Simple assignments: \( x := a \)
- Value of the variable on the left hand side takes the value given on the right hand side
- Can also implement nondeterministic assignments: \( x := \text{rand}(1, 10) \)

Multiple assignments: \( x, y := a, b \) or \( x := a || y := b \)
- Assign multiple variables at the same time (be careful not to confuse \( \| \) with \( \text{||} \) )

Guarded commands: \( g \rightarrow a \)
- Assignment (or “action”) is predicated on “guard”: only execute action if guard is true
- If the guard is true in a given state of the system, the guard is said to be “enabled”

Sequential composition: not formally implemented
- Unlike sequential programming languages, we will not assume sequential execution
- If you need to implement sequential computation, use a guarded commands + multiple assignments + a program counter (PC)

<table>
<thead>
<tr>
<th>Program</th>
<th>SequentialSwap</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>( x, y, \text{temp} : \text{int} ), ( pc : \text{nat} )</td>
</tr>
<tr>
<td>initially</td>
<td>( pc = 1 )</td>
</tr>
<tr>
<td>assign</td>
<td>( pc = 1 \rightarrow \text{temp}, pc := x, 2 )</td>
</tr>
<tr>
<td></td>
<td>( pc = 2 \rightarrow x, pc := y, 3 )</td>
</tr>
<tr>
<td></td>
<td>( pc = 3 \rightarrow y, pc := \text{temp}, 4 )</td>
</tr>
</tbody>
</table>
Example: Nondeterministic Door

Door dynamics: open and close at random

Person dynamics: move back and forth
- Can move back and forth between positions (p)
- Can only move from \( p = 0 \) to \( p = 1 \) if door is open

States: all possible values of variables
- Initial value marked by arrows

Actions: all possible transitions
- For guarded commands, guard must be true in order to execute the assignment
  \( \Rightarrow \) only include transition if guard is true
- Skip actions allow state to remain unchanged

Program: \textit{AutoDoor}

\begin{align*}
\text{var} & \quad d : \text{binary} \\
\text{initially} & \quad p = \{-1, 0, 1\} \\
\text{assign} & \quad d := 0 \\
& \quad d := 1 \\
& \quad p := 0 \quad \text{if} \quad p = -1 \\
& \quad p := -1 \quad \text{if} \quad p = 0 \\
& \quad p := 1 \quad \text{if} \quad (p = 0 \land d = 1) \\
& \quad p := 1 \quad \text{if} \quad p = 1 \\
\end{align*}
Program Execution: UNITY (Chandy and Misra)

UNITY = Unbounded Nondeterministic Iterative Transformations

Description

- **Program** consists of a set of (possibly guarded) variable assignments (or “actions”)
- **Behaviors** are generated by starting an initial state, then choosing any assignment for which the guard is true
- Command \((g \rightarrow a)\) may be evaluated in any order, at any time
- Require that all assignments be applied infinitely often in any execution (built in fairness)
- Reason about “programs” using formal (temporal) logic

Properties

- Useful for reasoning about systems in which there is very asynchronous behavior
- **Fairness** constraint is a bit too loose for some applications; only assume that each command executes eventually (instead of once every iteration) [more on this in a few slides]
Program Termination and Fixed Points

Q: Under the UNITY execution model, when is a program done (terminated)?
- Scenario #1: system might continue to go back and forth in a cycle
- Scenario #2: since the skip action is always enabled, we never really stop

A: P terminates at state v if any enabled action from v leaves the state unchanged
- We call such a state a Fixed Point (FP)

Simple example: what are the fixed points of the following programs?

Program
\[
\text{var } x, y : \text{number} \\
\text{assign} \\
x := y \\
\square y := f(7)
\]

Program
\[
\text{var } x, y : \text{number} \\
\text{assign} \\
x := y \\
\square x := 2 \\
y := f(7)
\]

Looking for fixed points on a program graph
- Let Reachable(V) represent the set of all vertices that can be reached (eventually) from a set of vertices V = \{v_1, v_2, \ldots, v_n\}
- A state v is a fixed point if Reachable({v}) = {v}
- A program may not terminate if the graph representing the program contains _____________
- For guarded program FP, all actions of the form g \rightarrow x := E must satisfy _____________
Distributed Systems

Distributed systems

- A distributed system consists of a set of agents (also called processes) and a set of directed channels.
- A channel is directed from one agent to one agent. The system can be represented by a directed graph (separate from the program graph within each agent).

Definition of the “state” of a distributed system

- Minimum amount of information such that the future behavior can be predicted without any other information about the past.
- Typically consists of the value of all variables that are part of any processes as well as messages that might be in transit.

Modeling a distributed system as a UNITY program

- Combine all variables from each agent + channel variables into list of variables for the (master) program.
- Combine all actions from each agent into actions for the program.
- Execute actions in arbitrary order.
Fairness

Weak Fairness
- Every action is guaranteed to be selected infinitely often
- Implication: between any two selections of a particular action, there are a finite (but unbounded) number of selections of other actions.

Strong Fairness
- Each action is selected infinitely often and if an action is enabled infinitely often then it is selected infinitely often
- Avoids situations where we get “unlucky” and never select an action at a time when it is enabled (mainly applies to guarded actions)

Door opening example
- Q: under weak fairness, does person always reach other side?
  \[
  \begin{align*}
  d &= 0 \\
  p &= 1 \\
  d &= 1 \\
  p &= 0 \\
  \end{align*}
  \]
- Q: what about under strong fairness?
  \[
  \begin{align*}
  d &= 0 \\
  p &= -1 ightarrow p := 0 \\
  d &= 1 \\
  p &= 0 ightarrow p = -1 \\
  d &= 1 \\
  p &= 0 ightarrow p = 1 \\
  \end{align*}
  \]
- Q: can you prove it?
Other Models of Scheduling

**UNITY**
Each command must be executed infinitely often.

**EPOCH**
Each command is executed before any are again.

**SYNCH(τ)**
In any interval, the difference in the number of times any two commands are executed is \( \leq \tau \).

\[
SYNCH(1) \subseteq EPOCH \subseteq SYNCH(2) \subseteq SYNCH(3) \subseteq \cdots \subseteq UNITY
\]

If program is correct for UNITY, it is correct for the others.
UNITY model provides (seemingly) simple description of programs
- Program = variables + actions [assignments] (that’s it!)
- Guarded assignment (g → a) allows modeling of finite state automata
- Distributed programs captured by nondeterministic execution model
- Termination = reaching a fixed point (variables remain constant)

Next: how to we prove that specifications are satisfied?
- A1: exhaustive testing [remember ZA002!]
- A2: model checking [for specific instantiation]
- A3: formal proof [often generalizable]

Fri: how to prove things using predicate calculus and quantification (review + some new stuff)

Next week: invariants (safety) and metrics (liveness)