Lecture 17: Nov 13
Fair Graphs

- Transition graphs + fairness constraints

- Operational semantics for fair modules

- Action of a graph $G = (\Sigma, \sigma^I, \rightarrow)$:
  a subset of $\rightarrow$

- Fairness constraint: pair of actions (α, β)

- Fairness Assumption: set of fairness constraints

- A fair graph $\mathcal{G}$ consists of a transition graph G and a fairness assumption F for G

Semantics of Fair Graphs

- For an action α, the ω-trajectory $\underline{s}$ is α-fair if $s_i \xrightarrow{\alpha} s_{i+1}$ for infinitely many i
- For a fairness constraint $f = (\alpha, \beta)$, $\underline{s}$ is f-fair if it is β-fair or not α-fair
  - Intuitively, if α repeats then so should β
- For a fairness assumption F, $\underline{s}$ is F-fair if it is f-fair for all $f \in F$
- The fair language $\mathcal{L}_\mathcal{G}$ of a fair graph $\mathcal{G}$ is the set of initialized F-fair trajectories
- Thm: The ω-language $\mathcal{L}_\mathcal{G}$ is a reactivity language

Restricted Types of Constraints

- Weak-fair constraint: $(\rightarrow, \alpha)$
  - $(\rightarrow, \alpha)$-fair trajectory means α-fair
  - Unconditional repetition
- Weak-fair graphs: all fairness constraints are weak
  - Easier to analyze
- Constraints specified by regions instead of actions
  - For a region σ, the ω-trajectory $\underline{s}$ is σ-fair if $s_i \in \sigma$ for infinitely many i
  - A fair graph with fairness assumption of the form $\{(\sigma_1, \tau_1), \ldots (\sigma_k, \tau_k)\}$ is called a Streett automaton
  - Weak-fair graph with single constraint specified by a region is called Büchi automaton
  - Büchi Automata to be studied later as a requirements language

Machine closed graphs

- A fair graph $\mathcal{G}$ is said to be machine-closed if every trajectory of $\mathcal{G}$ can be extended to a fair trajectory.

- Intuition: fairness assumption constrains only what happens in the limit, and cannot be violated afetr finitely many steps

- Analog of seriality for transition graphs: there is always a way to continue

- The set of prefixes of fair trajectories equals the set of (finite) trajectories

- How does it help: to verify safety properties, we can ignore fairness constraints

Local fairness

- Syntactic way to ensure machine closure
- A fairness constraint (α, β) is local if for all $s \stackrel{\alpha}{\rightarrow} t$, there is a state $u \in \Sigma$ such that $s \stackrel{\beta}{\rightarrow} u$.
  - whenever α is available so is β
- A locally-fair graph is a fair graph (G, F) such that
  - G is serial
  - F contains only local fairness constraints
- Every locally fair graph is machine closed

Execution of locally fair graphs

Input: a locally fair graph $\mathcal{G}$ and a state s;
Output: a source-s fair trajectory $\underline{s}$ of $\mathcal{G}$.

Queue: a queue of fairness constraints

Initialization
  $s_0 := s$;
  Queue equals F (in some order)

Update rounds.
for $i := 0$ to ∞ do
  Let Queue be $f_1 f_2 \ldots f_n$ with $f_k = (\alpha_k, \beta_k)$;
  if $Available(\alpha_k, s_i)$ for some $1 \leq k \leq n$
   then
    $j := min\ \{k \mid Available(\alpha_k, s_i)\}$;
    $s_{i+1} := Execute(\beta_j, s_i)$;
    $Queue := f_1 \ldots f_{j-1} f_{j+1} \ldots f_n f_j$
   else $s_{i+1} := Execute(\rightarrow, s_i)$

From Modules to Graphs

- With every update choice α, associate two actions:
  - Availability action: $avail_a$
  - Execution action: $exec_a$

- For every weak-fairness choice a, add the constraint

$$(\rightarrow_P, exec_a \cup (\rightarrow_P \setminus avail_a))$$

- For every strong-fairness choice a, add the constraint

$$(avail_a, exec_a)$$

- Every fair module $\mathcal{P}$ defines a fair graph $\mathcal{G}_\mathcal{P}$
- The resulting fair graph is local
- If no strong-fairness constraints, the resulting graph is weak-fair

Summary: Fairness

- We added fairness constraints for update choices in a module
  - Weak fairness: if a choice is continuously available then it must eventually be executed
  - Strong fairness: if a choice is repeatedly available then it must eventually be executed

- Operational semantics of fair modules: fair graphs

- Fair graphs have fairness constraints of the form (α, β): An infinite trajectory is (α, β)-fair if
  - either it contains only finitely many transitions in α
  - or infinitely many transitions in β

Fair Emptiness

- Given a graph G and a fairness assumption F, is there an initialized F-fair trajectory? (i.e. is the ω-language of the fair graph (G, F) empty?)

- Algorithms search for eventually periodic trajectories

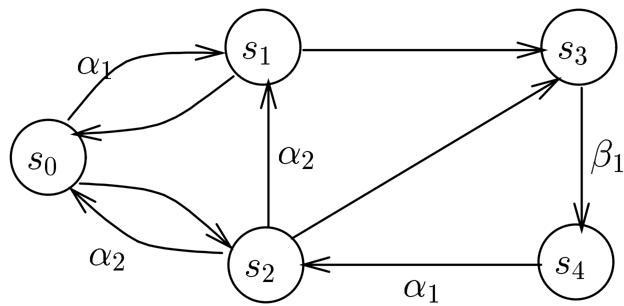
- Fair cycle problem:
  given a graph G and a fairness assumption F, is there a reachable fair cycle?

- Fair cycle: $s_0 \to s_1 \ldots \to s_m \to s_0$ such that for each $(\alpha, \beta) \in F$,
  - either it contains a transition from β,
  - or it does not contain a transition from α

- Computational problem: how to search for fair cycles?

Fair cycles: example

Two fairness constraints: $\{(\alpha_1, \beta_1), (\alpha_2, \beta_2)\}$



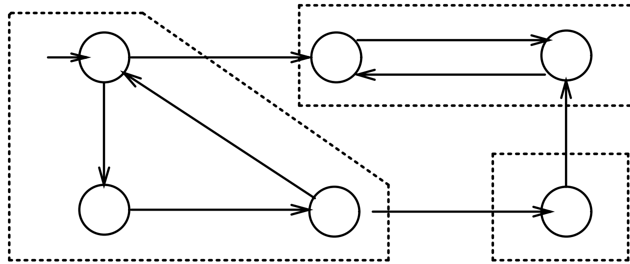
Fair Cycle Problem

- Input: Graph G and fairness assumption F
- Output: YES if there is a reachable fair cycle and NO otherwise
- Witness (debugging information)
  - Initialized trajectory leading to s_0
  - Fair cycle $s_0 s_1 \ldots s_m s_0$
- For finite graphs, there is a fair infinite trajectory iff there is a reachable fair cycle
- How do we search for fair cycles?
- Given a cycle, determining whether it is fair is easy, but we don't want to consider all cycles (exponentially many)

Strongly Connected Components

- Two states s and t are strongly connected if s is reachable from t and t is reachable from s
- A region σ is strongly connected if every pair of states in σ are strongly connected
- Strongly connected component: maximal strongly connected region
- For every graph $\cong_{scc}$ is the partition of its state-space into strongly connected components
- To search for fair cycles, we first compute strongly connected components

Example: strongly connected components



Strongly connected components can be computed in linear time by a modified depth-first search

Fair Components

- A strongly connected component σ is f-fair for $f = (\alpha, \beta)$ if it contains a f-fair cycle (i.e. a cycle that either contains β or avoids α)

- Fair component: a strongly connected component that is fair wrt all fairness constraints

- To solve fair cycle problem
  1. Compute $\cong_{scc}$ containing all SCCs
  2. Check if some $\sigma \in \cong_{scc}$ is fair

- Checking whether σ is f-fair: easy (linear)

- What if there are two constraints?
  - f_1-fair and f_2-fair does not mean $\{f_1, f_2\}$-fair

Handling multiple fairness constraints

- Let $F = \{(\alpha_1, \beta_1), \ldots (\alpha_k, \beta_k)\}$ and σ is one component
- If σ contains a transition from each β_i then there exists a F-fair cycle (why?)
- Suppose σ does not contain a transition from β_j. Then, every fair cycle must avoid α_j, so delete all the edges in α_j
- In general, delete all the edges in α_j for j such that σ does not intersect with β_j. These indices can be removed from F
- σ may no longer by strongly connected, so recompute the strongly connected components and repeat

Algorithm for Fair Components

```
function FSCC
Input: graph G and fairness assumption F
Output: YES if there is a F-fair SCC
foreach σ ∈ SCC(G) do
  →' := {(s,t) | s → t and s,t ∈ σ};
  F' := ∅;
  if →' ≠ ∅ then
    foreach (α,β) ∈ F do
      if →' ∩ β ≠ ∅
        then F' := Insert((α,β), F')
        else →' := →' \ α
        fi
      od;
    if F' = F then return YES
      else if FSCC((σ,σ,→'), F') = YES
        then return YES fi
      fi
  od;
return NO.
```

Complexity

- If F has ℓ constraints then at most ℓ recursive calls.
- Each call requires time $O((n+m)\cdot\ell)$.
- Overall complexity: $O((n+m)\cdot\ell^2)$.
- Special case: only weak constraints $(\rightarrow, \beta)$
  - No recursive calls
  - Complexity: $O((n+m)\cdot\ell)$.

Recurrence Verification

- An observation predicate p is a recurrent of of $\mathcal{P}$ if every fair ω-trajectory of $\mathcal{P}$ is p-fair
- A recurrent predicate is satisfied repeatedly on fair trajectories
- Recurrence verification problem:
  - Input: Fair module $\mathcal{P}$ and predicate p
  - Output: Is p is a recurrent of $\mathcal{P}$?
- E.g. Starvation freedom of mutual exclusion
  - Is $(pc_1 \neq req)$ a recurrent of $FairPete$

Solving Recurrence Verification

- Fair graph of fair module: (G,F)
- The predicate p is not recurrent if there is an ω-trajectory that is
  - F-fair and
  - not p-fair, i.e. $(\llbracket p \rrbracket, \emptyset)$-fair
- Consider the fair graph $(G, F \cup \{(\llbracket p \rrbracket, \emptyset)\})$
- Check if this fair graph has a reachable fair cycle

Response verification

- The predicate q is said to be a response to the predicate p in a fair module $\mathcal{P}$ if for every fair trajectory $\underline{s}$ of $\mathcal{P}$, for all $i \geq 0$, if $s_i \models p$ then for some $j \geq i$, $s_i \models q$.

- Intuitively, every request is followed by response

- Response verification:
  - Input: fair module $\mathcal{P}$, request predicate p, and response predicate q
  - Output: Is q a response to p in $\mathcal{P}$?

- Example: In Mutual exclusion,
  request: $pc_1 = req \lor pc_2 = req$
  response: $pc_1 = in \lor pc_2 = in$

- Recurrence verification is a special case:
  p is recurrent if p is a response to $true$

Solving Response Verification

- Reduce the response verification problem $(\mathcal{P}, p, q)$, to a recurrence verification problem by adding monitors (monitoring for liveness)
- Check if $alert = 0$ is recurrent of $\mathcal{P} \parallel ResponseMonitor$

module $ResponseMonitor$ **is**
  external p, q
  private $alert$: $bool$
  atom controls $alert$ **reads** p, q
    init
      $\| \ true \rightarrow alert' := 0$
    update
      $\| \ alert = 0 \wedge p \wedge \neg q \rightarrow alert' := 1$
      $\| \ alert = 1 \wedge q \rightarrow alert' := 0$

Summary: Solving Fair Cycle Problem

- General strategy: compute strongly-connected components, and check if one of them contains a fair cycle

- Recursive routine FSCC that calls SCC to compute strongly-connected components

- Complexity: $O((m+n)\ell^2)$

- Applications:
  - Recurrence verification: does every fair cycle of $\mathcal{P}$ contain a p-state?
  - Response verification: along every fair trajectory of $\mathcal{P}$, does every occurrence of request p followed by response q? (add monitor)

- When the module has only weak constraints, then improved algorithms are possible

Seaching for Büchi Cycle

- Büchi structure: single weak-fairness constraint specified by a region σ^T
- Büchi emptiness question:
  - Input: graph G and region σ^T
  - Is there a reachable cycle that contains some state in σ^T
- No need to compute SCCs
- Two stage solution:
  1. Primary search: Find out which states in σ^T are reachable from initial states of G
  2. Secondary search: For a reachable $s \in \sigma^T$, find out if s is reachable from itself
- For early termination interleave the two searches

Nested Search for Büchi Cycles

- DFS does primary search; visited states are stored in σ
- NDFS does secondary search for cycles; visited states are stored in τ
- The stack E contains states from which primary DFS is active (stack contains an initialized trajectory in reverse order)
- When $DFS(s)$ terminates, if s is in the fair region σ^T, invoke $NDFS(s)$
- If NDFS visits a state that is in the stack, it can conclude existence of a fair cycle

Main routine

Input: a finitely branching transition graph G,
and a finite region σ^T of G.
Output: Is there a reachable cycle that
intersects with σ^T?

```
input G: enumgraph;  σ^T: enumreg;
local σ, τ: enumreg;  E: stack of state;
   s: state
begin
   σ := EmptySet
   τ := EmptySet
   E := EmptyStack;
   foreach s in InitQueue(G) do
      if not IsMember(s, σ) then
         if DFS(s) then return YES fi;
         fi;
      od;
   return NO
   end.
```

Primary Search

```
function DFS: bool
input s: state;
local t: state;
begin
  E := Push(s, E)
  σ := Insert(s, σ);
  foreach t in PostQueue(s, G) do
    if not IsMember(t, σ) then
      if DFS(t) then return true
  if IsMember(s, σ^T) and not IsMember(s, τ)
    if NDFS(s) then return true
  E := Pop(E);
  return false
  end.
```

Secondary Search

```
function NDFS: bool
   input s: state;
   local t: state;
   begin
      τ := Insert(s, τ);
      foreach t in PostQueue(s, G) do
         if IsMember(t, E) then return true
         if not IsMember(s, τ) then
            if NDFS(t) then return true
      return false
      end.
```

Correctness

- If the algorithm returns true, then there is a fair cycle (easy)
- Suppose there is a fair cycle
- Order the states according the termination times of the primary search
- Let $s_0, \ldots s_k$ be the ordering of states in $\sigma^T \cap \sigma^R$ accordingly
- Let s_i be the first state in this list that belongs to a cycle
- Lemma: s_i does not belong to $post^*(s_j)$ for $j < i$
- Lemma: $NDFS(s_i)$ returns true

Complexity

- For every state s, DFS called with input s at most once, NDFS called with input s at most once
- Complexity: linear in the number of states and transitions
- Advantages over computing strongly-connected components
  - Graph does not have to be finite a priori
  - May terminate early looking at a fraction of the state space
  - Easy reporting of counter examples

Multiple Büchi Constraints

- Given $\sigma_1, \sigma_2, \ldots, \sigma_k$, we want to find if there is a reachable cycle that intersects with each of σ_i

- Graph with multiple constraints can be reduced to a graph with a single Büchi constraint

- Introduce a counter variable, initially 1.

- When the counter is i, if the current state belongs to σ_i, increment the counter to $i+1$ (modulo k), otherwise leave it unchanged

- Thm: Original graph has a fair cycle iff the transformed graph has a cycle in which counter is updated from k to 1.

- Similar trick works for fairness wrt a given set of actions

Recurrence verification of weak-fair modules

- Input: weak-fair module $\mathcal{P}$ and predicate p
- Goal: is there a reachable cycle that satisfies all fairness constraints of $\mathcal{P}$ and stays out of p-states

- Translate multiple weak-fair constraints to a single constraint
- During the secondary search do not explore p-states
- Optimizations
  - On-the-fly representation
  - Hashing