Lecture 7
Verification of Linear-Time Properties

Model checking Linear Temporal logic
Automata-Theoretic Approach [Vardi Wolper 1986]
• Question
  \( T |= \phi \)
• Construct
  Buchi Automaton
• Combine
  \( T \times A_{\neg \phi} \)
• Find
  accepting computation (error trace)

Translating formulas to Automata
\( \Box ( p \land \Box \neg q) \)

\( \varphi_1 \)
\( \varphi_2 \)

\( \varphi \)

\( p \land \neg \varphi \)

\( p \lor q \land \neg \varphi \)
Decomposing temporal operators

Pushing negations

\[ \neg \square \varphi \equiv \lozenge \neg \varphi \]
\[ \neg \lozenge \varphi \equiv \neg \square \varphi \]
\[ \neg ( \varphi \lor \psi ) \equiv \neg \varphi \lor \neg \psi \]
\[ \neg ( \varphi \land \psi ) \equiv \neg \varphi \land \neg \psi \]

Putting on Normal form

\[ \square \varphi \equiv \varphi \land \lozenge \square \varphi \]
\[ \lozenge \varphi \equiv \varphi \lor \lozenge \varphi \]
\[ \varphi \lor \psi \equiv (\varphi \land \lozenge \psi) \lor (\varphi \land \square \psi) \]
\[ \varphi \land \psi \equiv (\varphi \lor \lozenge \psi) \land (\varphi \lor \square \psi) \]

Translating formulas to Automata

\[ \varphi \equiv O( p \land \square \neg q ) \]

\[ \varphi \equiv O( p \land \square \neg q ) \equiv O( p \land \square \neg q ) \lor O( p \land \square \neg q ) \]

\[ \varphi \equiv O( p \land \square \neg q ) \equiv O( p \land \square \neg q ) \lor O( p \land \square \neg q ) \]

\[ \varphi \equiv O( p \land \square \neg q ) \equiv O( p \land \square \neg q ) \lor O( p \land \square \neg q ) \]

Translating formulas to Automata

\[ \varphi \equiv O( p \land \square \neg q ) \equiv O( p \land \square \neg q ) \lor O( p \land \square \neg q ) \]

\[ \varphi \equiv O( p \land \square \neg q ) \equiv O( p \land \square \neg q ) \lor O( p \land \square \neg q ) \]

Translating formulas to Automata

\[ \varphi \equiv O( p \land \square \neg q ) \equiv O( p \land \square \neg q ) \lor O( p \land \square \neg q ) \]

\[ \varphi \equiv O( p \land \square \neg q ) \equiv O( p \land \square \neg q ) \lor O( p \land \square \neg q ) \]

Translating formulas to Automata

\[ \varphi \equiv O( p \land \square \neg q ) \equiv O( p \land \square \neg q ) \lor O( p \land \square \neg q ) \]

\[ \varphi \equiv O( p \land \square \neg q ) \equiv O( p \land \square \neg q ) \lor O( p \land \square \neg q ) \]
Adding accepting states

\[ \varphi \equiv \Diamond (p \land \Box \neg q) \lor \Diamond (p \land \Box \neg q) \]

Searching for accepting computations

Safety properties:
- \( T \times A_{\neg q} \)
- Find a sequence of transitions from an initial state to a final states
- This is the reachability problem
- Can be solved by search from initial states
  - Visit all reachable states,
  - If accepting state is encountered, sequence of transitions = error trace

Liveness properties:
- Infinite computation of \( T \times A_{\neg q} \) must visit some accepting state infinitely many times
- This is the repeated reachability problem
- Can be solved by double search from initial states
  - Visit all reachable accepting states,
  - Search for loops from accepting states
  - If accepting loop ("lasso", "bad loop") is encountered = error trace

General technique for finding loops

- Accepting runs = strongly connected components reachable from initial state
  - containing at least one accepting state
  - Having at least one internal transition
- Finding all SCCs of a graph can be done by Tarjan’s algorithm (uses DFS)
- For finding some accepting cycle: more efficiently by nested depth-first search
  (Alur/Courcoubetis/Yannakakis, Holzmann)

Searching for reachable accepting loops

[See Courcoubetis 94]

1. Perform depth-first search of the reachable states
2. List reachable accepting states in post-order \( q_1, \ldots, q_n \)
3. Search from each \( q_i \) (starting with \( q_1 \)) to find a loop back to itself.
4. When post-order used, when searching states from \( q_j \), one need not reconsider states that were reached from states \( q_i \) with \( i < j \)

Why post-order is good

Post-order is order of popping states in DFS
If \( q_1, \ldots, q_n \) in post-order path from \( q_i \) to \( q_j \) with \( i < j \) must pass ancestor of \( q_i \)
Hence if state reached from \( q_i \) has path to \( q_j \) then \( q_i \) is in cycle

Why post-order is good

Post-order is order of popping states in DFS
If \( q_1, \ldots, q_n \) in post-order path from \( q_i \) to \( q_j \) with \( i < j \) must pass ancestor of \( q_i \)
Hence if state reached from \( q_i \) has path to \( q_j \) then \( q_i \) is in cycle
Why post-order is good

Post-order is order of popping states in DFS
If q1, ..., qn in post-order path from qi to qj with i < j must pass ancestor of qi

Hence if state reached from qi has path to qj then qi is in cycle

Pseudocode f. Nested Depth-First Search

Procedure DFS1(s):
  Stack.push(s); Visited.insert(s);
  ∀ successors s' of s do
    if s' not in Visited then DFS1(s');
    if accepting (s) then DFS2(s);
  Stack.pop();

Procedure DFS2(s):
  flagged(s) <- true;
  ∀ successors s' of s do
    if s' in Stack then return "exists cycle"
    if not flagged(s') then DFS2(s');

State-space exploration of PF

Reachable states

DFS tree

DFS tree with automaton

The graph has no loops that do not visit l3
Add self-loops at \( l_0 \) and \( m_0 \)

The graph has no loops that do not visit \( l_3 \)

Add self-loops at \( l_0 \) and \( m_0 \)

The graph has a bad loop that does not visit \( l_3 \)

Add self-loops at \( l_0 \) and \( m_0 \)

This is the first to be visited

Peterson-Fischer: Possible Specifications

Variables: \( y_1, y_2 \): boolean, \( t \): \( \{1,2\} \)
Initially \( y_1 = y_2 = false, \ t = 1 \)

Mutual Exclusion: \( \Box \sim (at \ l_3 \land at \ m_3) \)
Absence of Starvation: \( \Box (at \ l_2 \rightarrow \Diamond at \ l_3) \)
Bounded Overtaking: \( \Box (at \ l_2 \rightarrow \sim(at \ m_3 \ U (at \ m_3 \ U \sim(at \ m_3 \ U \sim(at \ l_3 \)))) \)

Simpler mutex

Variables: \( t \): \( \{1,2\} \)
Initially \( t = 1 \)

Mutual Exclusion: \( \Box \sim (at \ l_2 \land at \ m_3) \)
Absence of Starvation: \( \Box (at \ l_2 \rightarrow \Diamond at \ l_3) \)
Bounded Overtaking: \( \Box (at \ l_2 \rightarrow \sim(at \ m_3 \ U (at \ m_3 \ U \sim(at \ m_3 \ U \sim(at \ l_3 \)))) \)

Simpler state space
DFS tree

Bad states are unreachable

Reachable states
Accepting states for absence of starvation

Post-order

No loops can be found

Simpler mutex with self-loops

State space with self-loops

Variables: $t$: \{1,2\}
Initially $t = 1$

Mutual Exclusion: $\square \neg (at\ i2 \land at\ m3)$

Absence of Starvation: $\square \neg (at\ i2 \rightarrow \diamond at\ i3 )$

Bounded Overtaking: $\square (at\ i2 \rightarrow \neg (at\ m3 \cup (at\ m3 \cup \neg (at\ m3 \cup at\ i3 )))$
Peterson-Fischer with idle loops

Variables: $y_1, y_2$: boolean, $t$: (1,2)
Initially $y_1 = y_2 = false$, $t = 1$

We need mechanism to avoid computations that only perform idle steps

Fairness

- Assumption that some part of a transition system eventually progresses, without quantitative restrictions
- Can be viewed as an abstraction of many possible concrete transition scheduling policies.
- There are many different notions of fairness.
- The most common is weak fairness
- Another not uncommon is strong fairness

Fairness (definition)

- action $A$: any set of transitions (e.g., of one process)
- A computation $s_0\ s_1\ s_2\ s_3\ s_4\ s_5\ s_6\ s_7\ ...$ is
  - weakly fair wrp. to $A$ if
    - whenever $A$ is enabled in all $s_j$ with $j > i$
      - then some transition $A$ is taken from some $s_k$ with $k > i$
    - $□ (□ enabled A -> ◊ taken A)$
    - $◊□ enabled A -> □◊ taken A$
  - strongly fair wrp. to $A$ if
    - whenever $A$ is enabled in infinitely many $s_j$ with $j > i$
      - then some transition $A$ is taken from some $s_k$ with $k > i$
    - $□◊ enabled A -> ◊ taken A$
    - $◊□ enabled A -> □◊ taken A$
- Weakly fair wrp. to $A$ if
  - whenever $A$ is enabled in all $s_j$ with $j > i$
  - then some transition $A$ is taken from some $s_k$ with $k > i$
  - $□ (□ enabled A -> ◊ taken A)$
  - $◊□ enabled A -> □◊ taken A$
- Strongly fair wrp. to $A$ if
  - whenever $A$ is enabled in infinitely many $s_j$ with $j > i$
  - then some transition $A$ is taken from some $s_k$ with $k > i$
  - $□◊ enabled A -> ◊ taken A$
  - $◊□ enabled A -> □◊ taken A$
Checking fairness in exploration algorithm

To verify \( \phi \) under fairness assumption

- Naive algorithm searches for bad loops that satisfy fairness assumption \( \land \neg \phi \)

More efficient solution for weak fairness:

- search for bad loops that satisfy \( \neg \phi \)
  in which each action \( A \) with weak fairness is once either disabled or taken

Add self-loops at \( l_0 \) and \( m_0 \)

The loop does not have left process disabled or taken