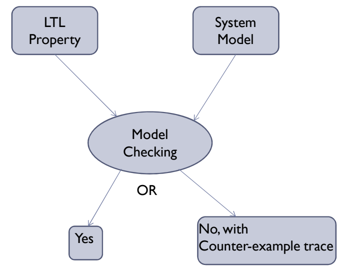
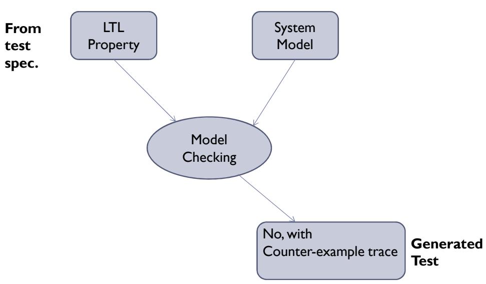
Model Checking
CS 4271, Lecture 4, 2^{nd} Feb 2012

Abhik Roychoudhury
http://www.comp.nus.edu.sg/~abhik

Model Checking



Recap: Model Checking for model-based testing



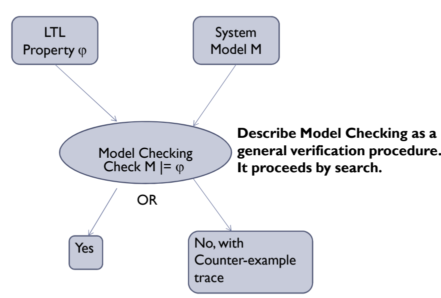
Encoding test specifications

- Def. 1
  - A trace σ satisfies a test specification M if σ contains at least one linearization of M as a **contiguous** subsequence.
  - Given MSC M,
    - define Lin(M) = set of linearizations of M.
    - For each linearization $\sigma = e_1, e_2, \ldots, e_k$ define
      - Define $prop_\sigma = \mathbf{F}(e_1 \wedge \mathbf{X}(e_2 \wedge \mathbf{X}(\ldots \mathbf{X}(e_k)\ldots)))$
    - Define property φ_M corresponding to M as
      - $\varphi_M = \neg (\vee_{\sigma \in Lin(M)} prop_\sigma)$
- A counter-example to φ_M is a test satisfying M.

Encoding test specifications

- Def. 2
  - A trace σ satisfies a test specification M if σ contains at least one linearization of M as a subsequence.
  - Given MSC M,
    - define Lin(M) = set of linearizations of M.
    - For each linearization $\sigma = e_1, e_2, \ldots, e_k$ define
      - $n_\sigma = \neg(e_1 \vee e_2 \vee \ldots \vee e_k)$
      - $prop_\sigma = (n_\sigma \mathbf{U} (e_1 \wedge \mathbf{X}(n_\sigma \mathbf{U}(e2 \wedge \mathbf{X}(\ldots \mathbf{X}(n_\sigma \mathbf{U} ek)\ldots))))$
    - Define property φ_M corresponding to M as
      - $\varphi_M = \neg (\vee_{\sigma \in Lin(M)} prop_\sigma)$
- A counter-example to φ_M is a test satisfying M.

Model Checking



LTL Model Checking – does M |= φ

1. Consider $\neg\varphi$. None of the exec. traces of M should satisfy $\neg\varphi$.
2. Construct a finite-state automata $A_{\neg\varphi}$ such that
   - Language($A_{\neg\varphi}$) = Traces satisfying $\neg\varphi$
3. Construct the synch product $M \times A_{\neg\varphi}$
4. Check whether any exec trace σ of M is an exec trace of the product $M \times A_{\neg\varphi}$ i.e. check Language($M \times A_{\neg\varphi}$) = empty-set?
   - Yes: Violation of φ found, report counterexample σ
   - No: Property φ holds for all exec traces of M.

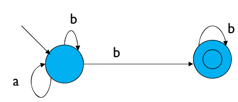
Recap: finite-state automata

- A = $(Q, \Sigma, Q_0, \to, F)$
  - Q is a finite set of states
  - Σ is a finite alphabet
  - $Q_0 \subseteq Q$ is the set of initial states
  - $\to \subseteq Q \times \Sigma \times Q$ is the transition relation
  - $F \subseteq Q$ is the set of final states.
- What is the language of such an automaton?

Recap: finite-state automata

- Regular languages:
  - Accept any finite-length string $\sigma \in \Sigma^*$ which ends in a final state.
- ω-regular languages:
  - Accept any infinite-length string $\sigma \in \Sigma^\omega$ which visits a final state infinitely many times.
- Set of strings accepted = *Language* of the automata.

Finite automata

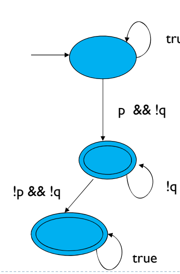


- Meaning as a regular language
  - $(a+b)^*b^+$
  - All finite length strings ending with b
- Meaning as a ω-regular language
  - All infinite length strings with finitely many a

LTL properties to automata

- Given a LTL property p
  - we want to convert p to an automata A_p such that
  - Language(A_p) = strings / traces satisfying p
- LTL properties are checked over infinite traces.
  - Given an infinite trace σ and a LTL property p, we can check whether σ |= p
- To convert LTL properties to finite-state automata, consider automata accepting infinite length traces.
  - Language(A_p) is ω-regular, not regular.

Example: LTL property to automata



Represents negation of the LTL property

G (p $\Rightarrow$ (p U q))

LTL properties to automata

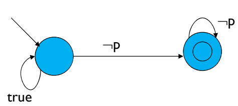
- Given a LTL property φ
  - We convert it to a ω-regular automata A_φ
- Language(A_φ) = {σ| σ∈Σ^ω ∧ σ |= φ}
  - Language(A_φ) is defined as per the ω-regular notion of string acceptance. It accepts infinite length strings.
  - All infinite length strings satisfying φ form the language of A_φ
  - Whether an infinite length string satisfies φ (or not) is defined as per LTL semantics.

Recall: LTL Model Checking

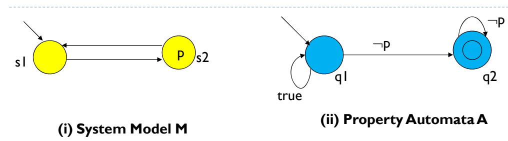
1. Consider ¬φ. None of the exec. traces of M should satisfy ¬φ.
2. Construct a finite-state automata $A_{\neg\varphi}$ such that
   - Language($A_{\neg\varphi}$) = Traces satisfying ¬φ
3. Construct the synch product M × $A_{\neg\varphi}$
4. Check whether any exec trace σ of M is an exec trace of the product M × $A_{\neg\varphi}$ i.e. check Language(M × $A_{\neg\varphi}$) = empty-set?
   - Yes:Violation of φ found, report counterexample σ
   - No: Property φ holds for all exec traces of M.

Example: Verify GFp

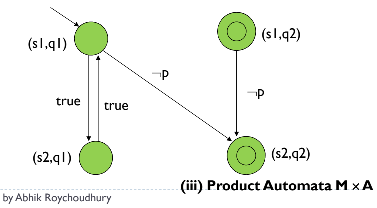
- Construct negation of the property
  - ¬GFp ≡ FG¬p
- Construct automata accepting infinite length traces satisfying FG¬p



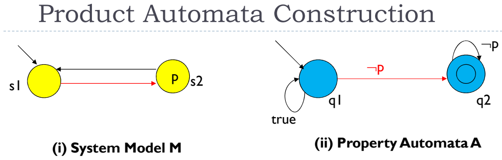
Product Automata



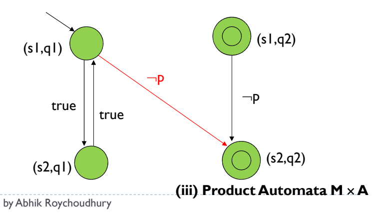
M |= GFp



Product Automata Construction

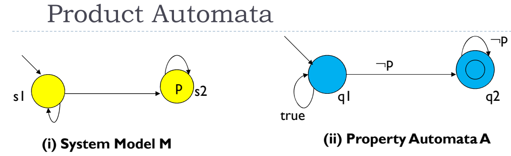


Note that s1 |= ¬p

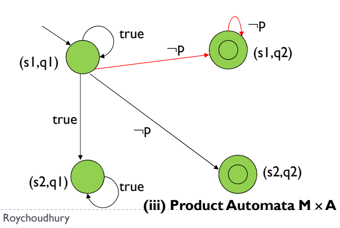


M |= GFp

Product Automata



M |≠ GF p

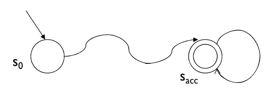


Recall: LTL Model Checking

1. Consider $\neg\varphi$. None of the exec. traces of M should satisfy $\neg\varphi$.
2. Construct a finite-state automata $A_{\neg\varphi}$ such that
   - Language($A_{\neg\varphi}$) = Traces satisfying $\neg\varphi$
3. Construct the synch product $M \times A_{\neg\varphi}$
4. Check whether any exec trace σ of M is an exec trace of the product $M \times A_{\neg\varphi}$ i.e. check Language($M \times A_{\neg\varphi}$) = empty-set?
   - Yes: Violation of φ found, report counterexample σ
   - No: Property φ holds for all exec traces of M.

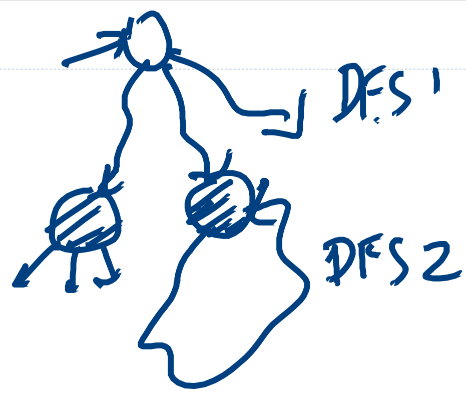
Emptiness Check

- Language($M \times A_{\neg\varphi}$) = empty-set?
  - Is there any trace which visits one of the accepting states of the product automata infinitely many times?
  - Look for accepting cycles.



Emptiness Check

- Perform DFS from initial state until you reach an accepting state s_{acc}
- When you reach s_{acc}, remember s_{acc} in a global var. and start a nested DFS from s_{acc}
  - Stop the nested DFS if you can reach s_{acc}
- If no accepting cycles are found, report yes.
- If accepting cycles are found
  - Concatenate the two DFS stacks and report it as counter-example trace of the LTL property.
- This algo. is implemented in SPIN model checker.



Nested DFS – step 1

- procedure dfs1(s)
  - push s to Stack1
  - add {s} to States1
  - if accepting(s) then
  - States2 := empty; seed := s; dfs2(s)
  - endif
  - for each transition s $\rightarrow$ s' do
  - if s' $\notin$ States1 then df1(s')
  - endfor
  - pop s from Stack1
- end

Nested DFS – step 2

- procedure dfs2(s)
  - push s to Stack2
  - add {s} to States2
  - for each transition s $\rightarrow$ s' do
  - if s' = seed then report acceptance cycle
  - else if s' $\notin$ States2 then df2(s')
  - endif
  - endfor
  - pop s from Stack2
- end

