EECS 219C: Computer-Aided Verification

Properties as Automata and Explicit-State Model Checking

Sanjit A. Seshia

EECS, UC Berkeley

Announcements

- HW 1 due on Wednesday
- Make-up class on Friday, 2/23
  - 540 Cory
  - 11 am - 12:30 pm
- Project topics due tonight
  - proposals due Feb. 21

Today's Lecture

- Recap of Models, Temporal Logic
  - Temporal logic and Automata
- Explicit-state model checking
  - Search algorithms: DFS, BFS
  - Verifying safety and liveness
  - Optimizations

Recap

- Models
  - Closed systems
  - Kripke structures (S, S_0, R, L)
    - L is a labeling function, mapping a state to a set of atomic propositions (Boolean formulas) true in that state
- Properties
  - Temporal logic (LTL, CTL)

More on Models

- Typically the overall system is specified as a set of modules, and the environment
  – Assume we have a Kripke structure for each
- There are two ways of constructing the overall Kripke structure
  – Synchronous composition
  – Asynchronous composition

Synchronous Product

- Given two Kripke structures
  – $M1 = (S1, s1_0, R1, L1)$
  – $M2 = (S2, s2_0, R2, L2)$
- Sync. Product is $M = (S, s_0, R, L)$
  – $S \subseteq S1 \times S2$
  – $s_0 = (s1_0, s2_0)$
  – $R = R1 \wedge R2$
  – $L(s1, s2) = (L1(s1), L2(s2))$

Asynchronous Product

- Given two Kripke structures
  - $M1 = (S1, s1_0, R1, L1)$
  - $M2 = (S2, s2_0, R2, L2)$
- Async. Product is $M = (S, s_0, R, L)$
  - $S \subseteq S1 \times S2$
  - $s_0 = (s1_0, s2_0)$
  - $R(s) = (R1(s1,s1') \wedge s2' = s2)$
    $\vee (R2(s2,s2') \wedge s1' = s1)$
  - $L(s1, s2) = (L1(s1), L2(s2))$

Some Remarks on Temporal Logic

- The vast majority of properties are safety properties
- Liveness properties are useful abstractions of more complicated safety properties (such as real-time response constraints)

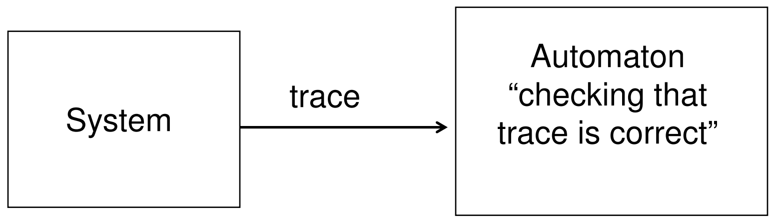
Deadlock

- An oft-cited property, especially people building distributed / concurrent systems
- Can you express it in terms of
  - a property of the state graph?
  - a CTL property?
  - a LTL property?

Next

- Connections between temporal logic and automata

Mental Picture



Automata from Kripke Structures

- Recall: Trace is a sequence of the observable parts of states (labels)
- Each label is a set of atomic propositions, but can be thought of as a symbol in an alphabet
  - Alphabet is 2^{AP}, where AP is set of atomic propositions
- Now we can talk about automata that accept traces

Recap: Automata over Finite Traces

- Just your regular finite automaton with an accepting state
  - All finite traces (words) that take the automaton into the accepting state are "in its language"
- But behaviors (and traces) are infinite length
  - So we need a new notion of acceptance

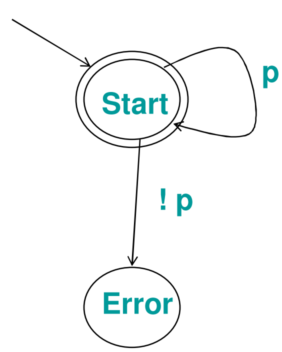
Automata over Infinite Traces

- What does "Accept" mean?
  - Certain states of the automaton are called "accepting states"
  - At least one accepting state must be visited infinitely often
- Such automata are called Büchi automata
  - Also Omega-automata (written ω-automata)

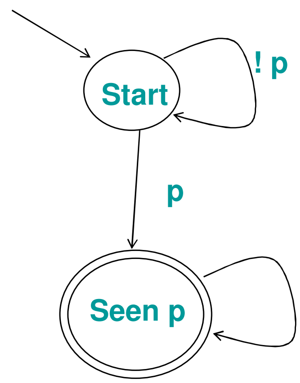
From Temporal Logic to Automata

- Properties are often specified as automata
- A (Buchi) automaton corresponding to a temporal logic formula ϕ *accepts* exactly those traces that satisfy ϕ

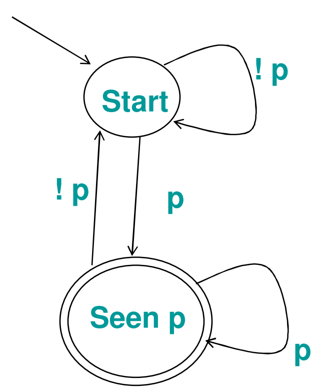
Automaton for G p, p a Boolean formula



Automaton for F p



Automaton for GFp



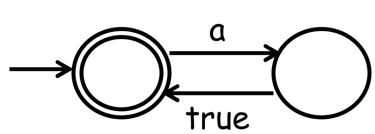
From LTL to Automata

- Any LTL formula can be translated to a corresponding automaton
- There are many translation algorithms
  - We won't do any in class
- How about the other way around?
  - Can an arbitrary Buchi automaton be translated into an LTL formula?

Automaton without LTL counterpart

Automata are more expressive than LTL

What traces does the automaton below accept?

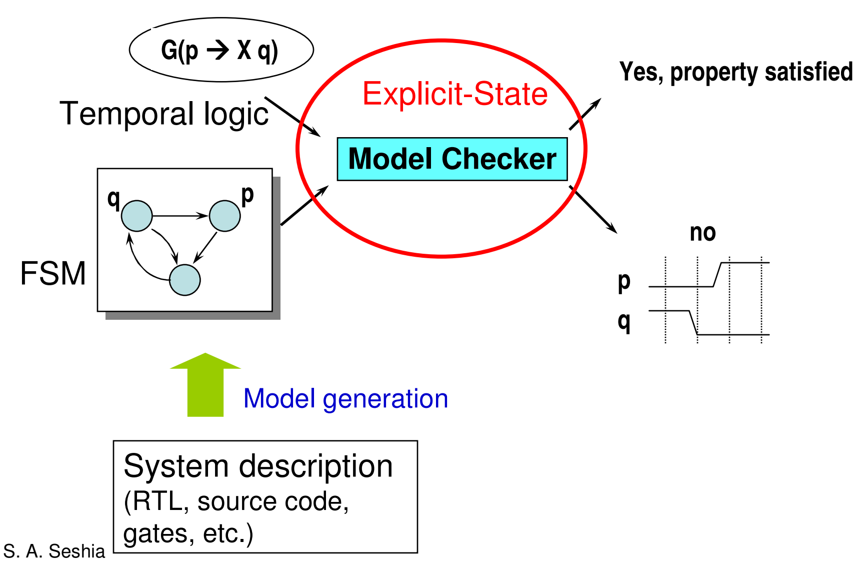


Claim: This cannot be expressed in LTL.

(How about $a \wedge G\ (a \Rightarrow X\ X\ a)$?)

On to Model Checking …

Finite-State Model Checking



Explicit-State Model Checking

- Model checking exhaustively enumerates the states of the system
- State space can be viewed as a graph
- Explicit-state model checking
  - Explicitly enumerates each state and traverses each edge of the graph
- We will focus on explicit-state techniques as used in SPIN [G. Holzmann, won ACM Software Systems Award]

Issues with Explicit-State MC

- The graph is usually HUGE (> 10^6 nodes)
  - So can't compute it a-priori
- But we are given an initial state (s_0) and a way of going from state to state (transition relation R)
  - In particular, we'll assume that R is specified as a "set of actions", each having a "enabling condition" and a "set of assignments" that cause a state change

Model Checking G p

- Consider the simplest property G p
  - p is a system invariant to be satisfied by all states
- Given the state graph, how can we check this?

Model Checking G p

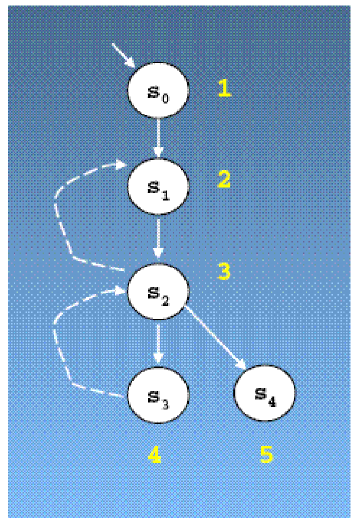
- Consider the simplest property G p
  - p is a system invariant to be satisfied by all states
- Given the state graph, how can we check this?
  - Graph traversal: DFS or BFS

Depth-First Search (DFS)

Maintain 2 data structures:

1. Set of visited states
2. Stack with current path from the initial state

Potential problems?



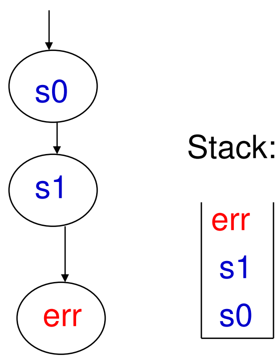
Generating counterexamples

If the DFS algorithm finds an “error” state (in which p is not satisfied), how can we generate a counterexample trace from the initial state to that state?

Generating counterexamples

If the DFS algorithm finds an “error” state (in which p is not satisfied), how can we generate a counterexample trace from the initial state to that state?

Will this be the shortest counterexample?



DFS without State Set

- Only keep track of current stack
- No set of states to maintain
  - Each time you visit a state, check whether it’s on the stack
    - If so, don’t explore its edges
    - If not, do.
- Q1: Will this terminate?
- Q2: If yes: on state graph with n states, how long will it take?

Bounded Model Checking with DFS

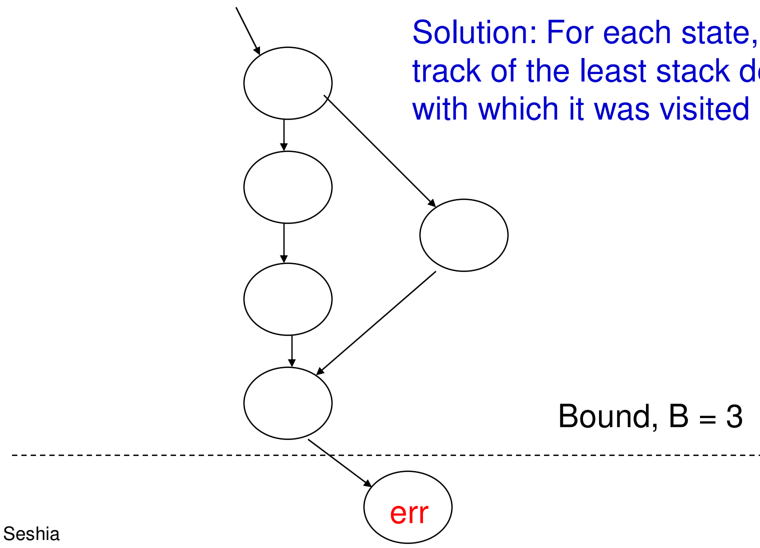
- Same as the original DFS, except that you only allow your stack to grow up to B elements deep
  - Keep track of set of all visited states and explore a state only if it is not in this set
- If this returns “no error within B steps from initial state”, can you trust it?

Bounded Model Checking with DFS

- Same as the original DFS, except that you only allow your stack to grow up to B elements deep
  - Keep track of set of all visited states and explore a state only if it is not in this set
- If this returns “no error within B steps from initial state”, can you trust it?

  - NO! Example on next slide

Example



Solution: For each state, keep track of the least stack depth with which it was visited

Breadth-First Search

- Visit states in order of distance from initial state
- Uses queue, No stack: how to generate counterexamples?
- Are the generated counterexamples the shortest?

Comparing DFS and BFS for Gp

- Pros of BFS over DFS
  - Shortest counterexample generated
- Cons of BFS
  - Need to store back-pointers to predecessor with each state in the state space representation (increased memory requirement)
  - Does not efficiently extend to liveness properties
    - Need to do cycle detection

What about non-Gp safety properties?

- Recall: safety properties → finite counterexample trace
- So we can construct a monitor automaton with an "error" state that must be avoided
  - Construct product of that automaton with original system
  - Error state of product has "error" in the component corresponding to the monitor