EECS 219C: Formal Methods

Explicit-State Model Checking: Additional Material

Sanjit A. Seshia

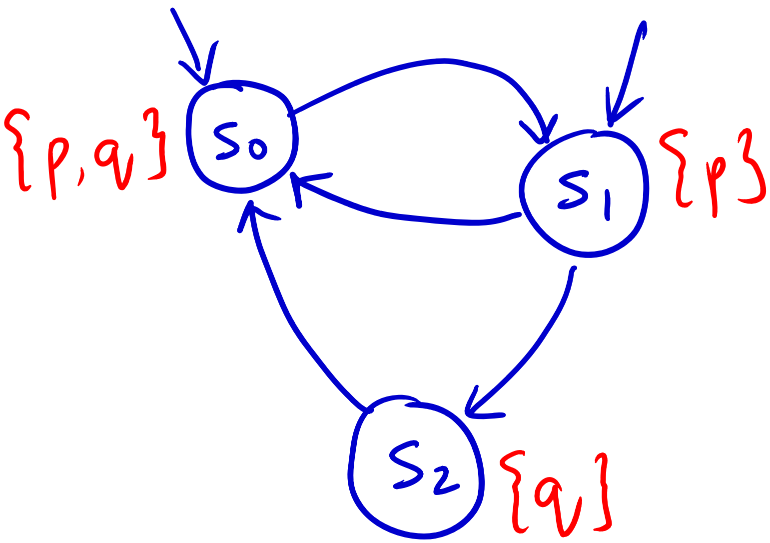
EECS, UC Berkeley

Acknowledgments: G. Holzmann

Checking if M satisfies ϕ: Steps

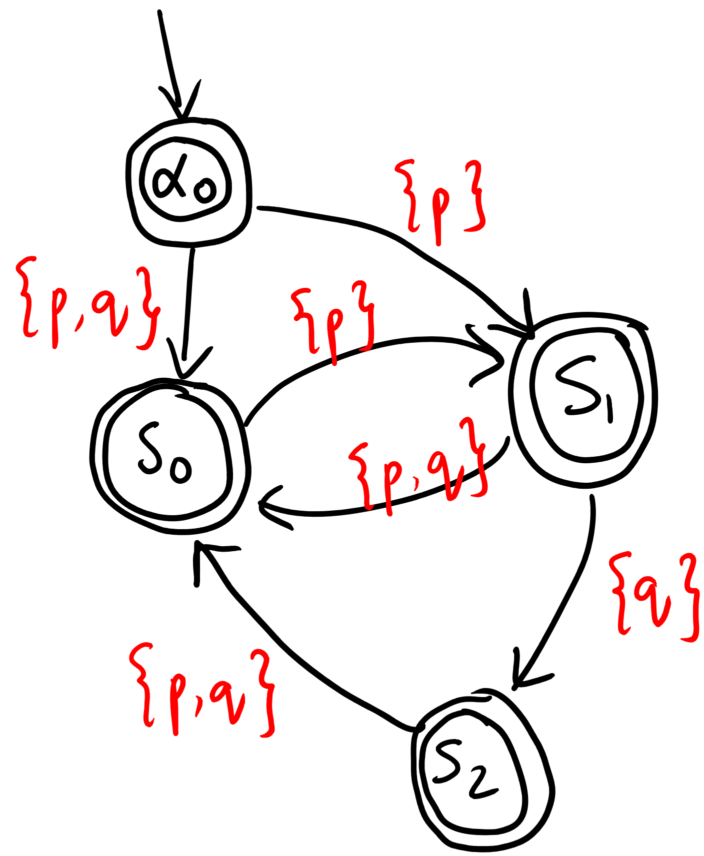
1. Compute Buchi automaton B corresponding to $\sim\phi$
2. Compute the Buchi automaton A corresponding to the system M
3. Compute the *synchronous* product P of A and B
   - Product computation defines “accepting” states of P based on those of B
4. Check if some “accepting” state of P is visited infinitely often
   - If so: we found a bug
   - If not, no bug in M

Example of Step 2



Kripke structure

What's different between the two? What's the same?



Corresponding Buchi automaton (transitions on labels not shown go to a non-accepting sink state "err")

Step 1: Buchi Automaton from Kripke Structure

- Given: Kripke structure $M = (S, S_0, R, L)$
  - $L : S \rightarrow 2^{AP}$, AP – set of atomic propositions
- Construct Buchi automaton $A = (\Sigma, S \cup \{\alpha_0, err\}, \Delta, \{\alpha_0\}, S \cup \{\alpha_0\})$ where:
  - Alphabet, $\Sigma = 2^{AP}$
  - Set of states = $S \cup \{\alpha_0, err\}$
    - α_0 is a special start state, err is a (sink) error state
  - All states are accepting except err
  - Δ is transition relation of A such that:
    - $\Delta(s, \sigma, s')$ iff $R(s, s')$ and $\sigma = L(s')$
    - $\Delta(\alpha_0, \sigma, s)$ iff $s \in S_0$ and $\sigma = L(s)$

Need to also add transitions to dummy error state err for other symbols σ not covered above

Step 2: Compute synchronous product of A with B

- A and B are both Buchi automata with the same alphabet
- Synchronous product:
  - $A = (\Sigma, S_1, \Delta_1, \{s_0\}, S_1 \backslash \{err\})$ (err is dummy error state)
  - $B = (\Sigma, S_2, \Delta_2, \{s_0'\}, F')$
  - Product $P = (\Sigma, S_1 \times S_2, \Delta, \{s_0, s_0'\}, F)$
    - $\Delta((s_1, s_2), \sigma, (s_1', s_2')) = \Delta_1 (s_1, \sigma, s_1') \wedge \Delta_2 (s_2, \sigma, s_2')$
    - $(s_1, s_2) \in F$ iff $s_1 \neq err \wedge s_2 \in F'$

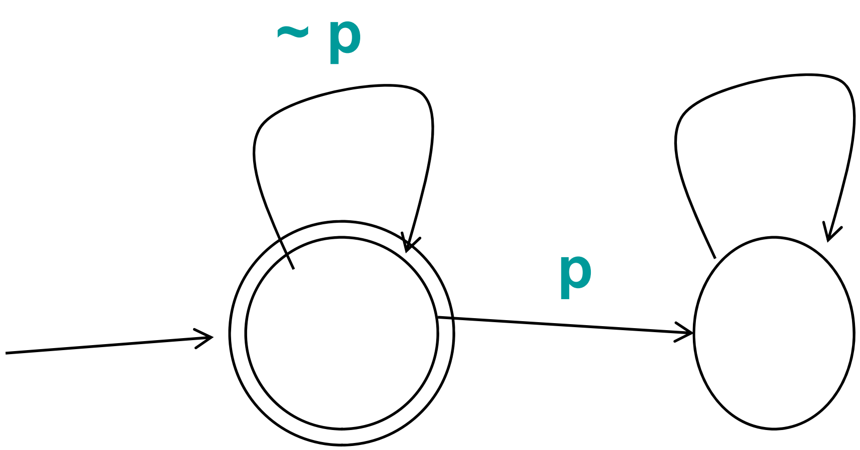
[Note: we assume above that they share the same alphabet Σ]

Example of Step 2

Property ϕ: **F** q

Step 3: Checking if some state is visited infinitely often

- Suppose I show you the graph corresponding to the product automaton
- What graph property corresponds to "visited infinitely often"?



Step 3: Checking if some state is visited infinitely often

- Suppose I show you the graph corresponding to the product automaton
- What graph property corresponds to "visited infinitely often"?
  - Checking for a cycle with an accepting state
  - We also need to check that the accepting state is reachable from the initial state

DFS + cycle detection

- How can we modify DFS to do cycle detection?

DFS + cycle detection

- How can we modify DFS to do cycle detection?
  - Find strongly connected components, and then check if there's one with an accepting state
    [But: we don't have the graph with us to start with]
  - Use DFS to find an accepting state s
    - On finding one, explore its child nodes.
    - If a child node is on the stack, or if s has a self loop, we're done [Easy to see why]
    - Else, do a new DFS starting from s to see if you can reach it again [Why will this work? Any modifications to the basic DFS needed?]
    - SPIN's "nested DFS" algorithm

Checking if M satisfies ϕ: Steps

1. Compute Buchi automaton B corresponding to $\sim\phi$
2. Compute the Buchi automaton A corresponding to the system M
3. Compute the *synchronous* product P of A and B
   - Product computation defines “accepting” states of P based on those of B
4. Check if some “accepting” state of P is visited infinitely often
   - If so: we found a bug (What does a counterexample look like?)
   - If not, no bug in M

What if our property is not LTL?

- Let’s say the property is specified directly as a Buchi automaton B
- Then, to check if the system A satisfies the property, we use the same algorithm as before:
  – Compute complement of B: call it B’
  – Compute sync. product of A and B’
  – Check for loops involving “accepting” states
- IMP: Buchi automata are closed under complementation, union, intersection
- Nondeterministic Buchi automata are strictly more expressive than deterministic Buchi automata!

Time/Space Complexity

- Size measured in terms of:
  - N_A – num of states in system automaton
  - N_B – num of states in property automaton (for complement of the property we want to prove)
  - N_S – num of bits to represent each state
  - N_E – num transitions in product automaton
  - Total size = $N = (N_A * N_B * N_S) + N_E$
- Checking G p properties w/ DFS
  - Time: ? Space: ?
- Checking arbitrary (liveness) properties w/ nested DFS
  - Time: ? Space: ?

Time/Space Complexity

- Size measured in terms of:
  - N_A – num of states in system automaton
  - N_B – num of states in property automaton (for complement of the property we want to prove)
  - N_S – num of bits to represent each state
  - N_E – num transitions in product automaton
  - Total size = $N = (N_A * N_B * N_S) + N_E$
- Checking G p properties w/ DFS
  - Time: O(N*L) [X] Space: O(N) {L – lookup time to check if state visited already}
- Checking arbitrary (liveness) properties w/ nested DFS
  - Time: O(N*L) [2X] Space: O(N)

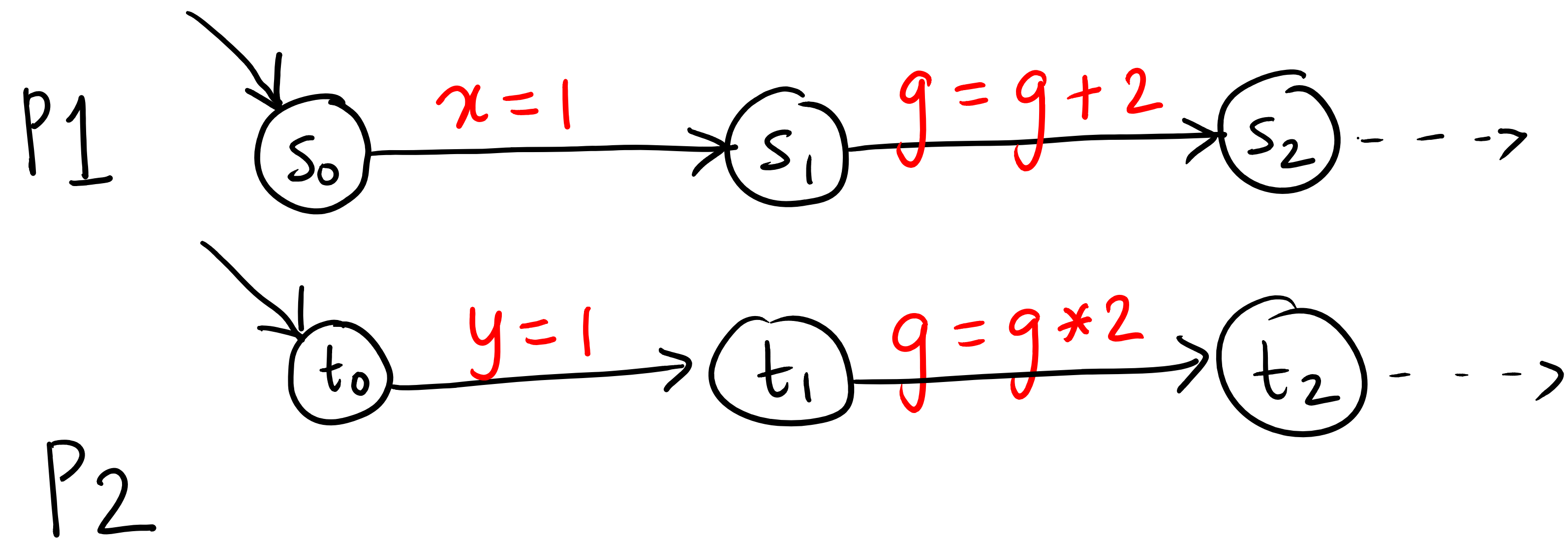
Optimizations

- Complexity is a function of $N_E + N_A * N_B * N_S$
- Natural strategy to reduce time/space is to reduce:
  - N_E, N_A → Partial-order reduction, Abstraction (later lecture)
  - N_B → not really needed, N_B is usually small
  - N_S → State compression techniques

Partial Order Reduction

- Edges of automata correspond to “actions” taken by the automaton
  - Assume that you label each edge with its corresponding action
- Idea: Some actions are independent of each other
  - E.g. “internal actions” of systems composed asynchronously
  - You can permute them without changing the end state reached
    - Both interleavings yield same end state

An Example



Initial state: x = y = g = 0

Starting in (s_0, t_0), what are the possible executions?

Some Sample Properties: Are they preserved by P-O Reduction?

- F (g ¸ 2)

- G (x ¸ y)

Key point: The property matters in deciding dependencies!

Atomic propositions that appear in the temporal logic property are termed "**relevant atomic propositions**"

Implementing P-O Reduction

- At each state s, some set of actions is enabled: enabled(s)
- Of this set, we want to explore only a subset ample(s) s.t.
  - We explore a subset of states and transitions
  - The property holds for the reduced system iff it holds for the full system
- Pick an arbitrary element of ample(s) and execute that action
- QN: How to compute ample(s)?

Independence and Invisibility

- Important properties of actions a, b: **independence** & **invisibility**
- Independence
  - Enabledness: Action a should not disable b, and vice-versa
  - Commutativity: a(b(s)) = b(a(s))
- Invisibility
  - a and b should not affect the values of any 'relevant' atomic propositions in the LTL property

Problem

- Computing ample(s) exactly is as hard as computing the reachable states of the system!
  - One of the conditions defining ample(s): Along every path starting at s, an action *a* dependent on action *b* in ample(s) cannot be executed before *b*
- See [Ch. 10, Clarke,Grumberg,Peled] for a proof

Computing ample(s)

- Conservative heuristics to compute actions that are NOT in ample(s):
  - ample(s) cannot have actions that are visible or dependent on other actions in enabled(s)

1. If the same variable appears in two actions, they are dependent
2. If two actions appear in the same process/module, they are dependent
3. If an action shares a variable with a relevant atomic proposition, then it is visible

Summary of P-O Reduction

- Very effective for asynchronous systems
- SPIN uses it by default

State Compression Techniques

- Lossless
  - Collapse compaction
    - Essential a state encoding method
- Lossy
  - Hash compaction
    - Replace state vector by its hash; if you visit a state with same hash as previously visited, then don't explore further
  - Bit-state hashing
    - Think of the hash as a memory address of a single bit that represents whether the state has/hasn't been visited
    - SPIN uses multiple (2) hashes per state
    - 500 MB of memory can store 2 . 10^9 states with 2 hashes
  - Are errors found this way still valid errors?
  - Often even if a state is missed, its successors are reached