EECS 219C: Computer-Aided Verification

Explicit-State Model Checking: Liveness and Optimizations

Sanjit A. Seshia

EECS, UC Berkeley

Thanks to G. Holzmann

Deadlock

- Any insights on how to specify deadlock?

Deadlock

- Some observations
  - OS textbook: by Silberschatz, Galvin, … defines deadlock-freedom in a way that be written as a "G p" property
- But "natural" way of defining it is as a liveness property

  AG EF ("make progress")

S. A. Seshia 3

Today's Lecture

- Explicit-state model checking
  - Verifying liveness
  - Optimizations needed to make it work in practice

S. A. Seshia 4

Focus on Asynchronous Systems

- Today's lecture will focus on asynchronous systems
- This is what SPIN is targeted towards
  - Key optimizations in SPIN make use of the asynchronous composition of systems
  - However, synchronous composition has one important use too

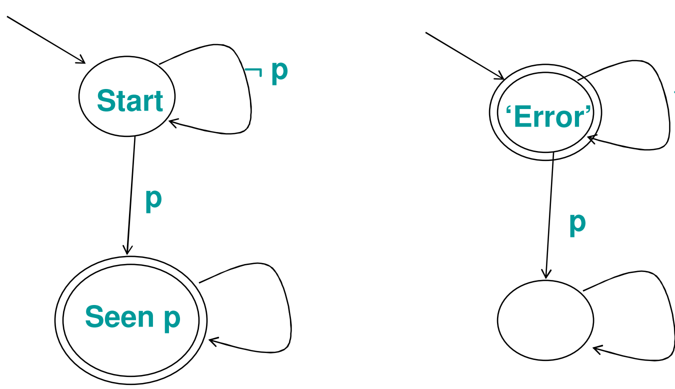
Recap: Checking G p

- Explore states and check that each one satisfies p
  - Alternatively check that none satisfy $\neg$ p
- This works for safety properties that are properties of a single "state"
  - Deadlock could be characterized this way if defined as a safety property
- Need something different for general properties

Properties and Automata

- Every LTL property has a corresponding Buchi automaton
- Given a “good” property ϕ that you want to prove, its negation is a “bad” property ϕ’ that the system should not satisfy
  - ϕ’ has a corresponding Buchi automaton B’ too
  - Error conditions indicated by visiting “accepting states” of B’ infinitely often
- If the system M satisfies ϕ’, it means that M has a bug, otherwise, it’s correct

Example: Automata for F p & G ($\neg$p)



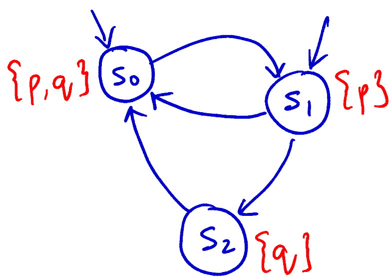
Checking Arbitrary LTL

- Given:
  - Kripke structure for system, M
  - Buchi automata for negation of LTL property, B'
- How do we check if M satisfies B' (and hence has a bug)?

Checking if M satisfies B': Steps

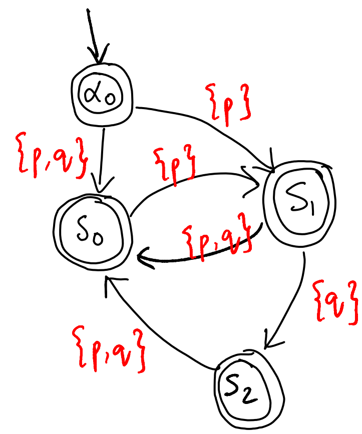
1. Compute the Buchi automaton A corresponding to the system M
2. Compute the *synchronous* product P of A and B'
   - Product computation defines "accepting" states of P based on those of B'
3. Check if some "accepting" state of P is visited infinitely often
   - If so: we found a bug
   - If not, no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

Example of Step 1



Kripke structure

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



Corresponding Buchi automaton

Step 1: Buchi Automaton from Kripke Structure

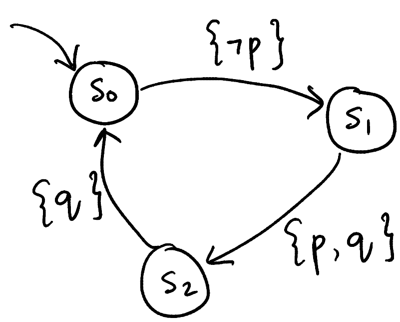
- Given: Kripke structure $M = (S, S_0, R, L)$
  - $L : S \rightarrow 2^{AP}$, AP – set of atomic propositions
- Construct Buchi automaton $A = (\Sigma, S \cup \{\alpha_0\}, \Delta, \{\alpha_0\}, S \cup \{\alpha_0\})$ where:
  - Alphabet, $\Sigma = 2^{AP}$
  - Set of states = $S \cup \{\alpha_0\}$
    - α_0 is a special start state
  - All states are accepting
  - 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 $\sigma = L(s)$

Step 2: Compute synchronous product of A with B'

- A and B' are both Buchi automata with the same alphabet
- Synchronous product:
  - $A = (\Sigma, S_1, \Delta_1, \{s_0\}, S_1)$
  - $B' = (\Sigma, S_2, \Delta_2, \{s_0'\}, F')$
  - 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 $s_2 \in F'$ (i.e., an accepting state is defined by an accepting state of B')

Example of Step 2

- Compute product of this example automaton A with that for $G \neg p$



(all states are accepting)

Note that the labels in the property automaton are to be interpreted differently from those in A

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?

S. A. Seshia 17

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 [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

S. A. Seshia 18

Checking if M satisfies B': Steps

1. Compute the Buchi automaton A corresponding to the system M
2. Compute the *synchronous* product P of A and B'
   - Product computation defines "accepting" states of P based on those of B'
3. 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

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
  - Total size = $N = N_A * N_B * N_S$
- 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
  - Total size = $N = N_A * N_B * N_S$
- 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)

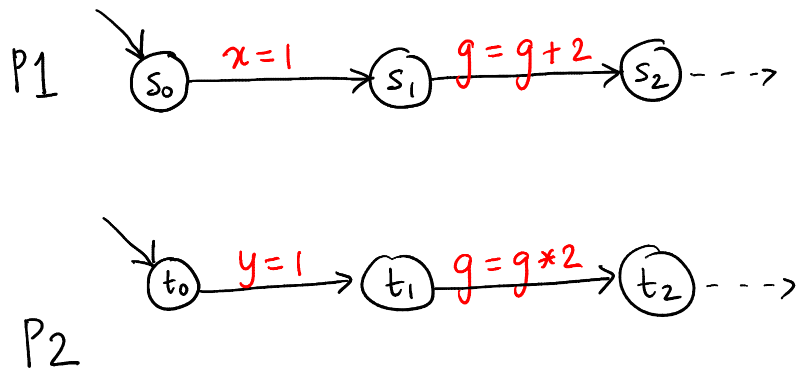
Optimizations

- Complexity is a function of N_A * N_B * N_S
- Natural strategy to reduce time/space is to reduce:
  - 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

- Labels on edges of automata can be thought of as "actions"
  - An action for an edge sets the proposition labeling that edge to true
  - Often these actions are "internal actions" of systems composed asynchronously
- Idea: Some actions are independent of each other
  - You can permute them without changing the end state reached
    - Both interleavings yield same end state

An Example



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

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

- F $(g \geq 2)$
- G $(x \geq y)$

Key point: The property matters in deciding dependencies!

Implementing P-O Reduction

- At each state s, some set of actions is enabled: enabled(s)
- Of this set, a subset are such that any interleaving of them yields the same end state and they do not "influence" other actions: ample(s)
  - Pick one order for elements of ample(s) and execute all those actions first in that order
- QN: How to compute ample(s)?

Computing ample(s)

- Important characteristics of elements a, b of ample(s): must be independent & invisible
  - Action a should not disable b, and vice-versa
  - The effect of ample(s) actions should not affect the values of any 'relevant' atomic propositions in the LTL property
- Conservative heuristics to compute ample(s):
  - If the same variable appears in two actions, they are dependent
  - If two actions appear in the same process/module, they are dependent
  - 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

S. A. Seshia 29

State Compression Techniques

- Lossless
  - Collapse compaction
    - Essential a state encoding method
- Lossy (sacrifice completeness!)
  - Hash compaction
    - Replace state vector by its hash; if you visit a state with same hash as previously visited, then what?
  - 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

S. A. Seshia 30

Next class

- Basic concepts for symbolic model checking
  - Start μ-calculus, QBF, etc.