Overview

- Automata-Theoretic Model-Checking
  - Automata on finite and infinite words
  - Representing models and formulas
  - Model checking using automata
  - Partial order reduction and closure under stuttering
- Implementing automata-theoretic model checking
  - Checking emptiness
  - Nested DFS
  - Bitstate hashing
- SPIN/Promela
  - expressing models in Promela
  - using SPIN

Automata-Theoretic LTL Model-Checking

Marsha Chechik

University of Toronto

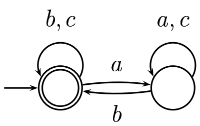
Automata on Finite Words, Cont'd

Let v be a word of Σ^* of length $|v|$. A *run* of $\mathcal{A}$ over v is a mapping $\rho : \{0, 1, ..., |v|\} \to Q$ s.t.

- First state is the initial state: $\rho(0) \in Q^0$
- $\forall 0 \leq i \leq |v| \cdot (\rho(i), v(i), \rho(i+1)) \in \Delta$

A run ρ of $\mathcal{A}$ on v – a path in automaton to a state $\rho(|v|)$ where the edges are labeled with letters in v (so v is *input* to $\mathcal{A}$).

A run is *accepting* if $\rho(|v|) \in F$. An automaton $\mathcal{A}$ accepts a word v iff exists an accepting run of $\mathcal{A}$ on v.

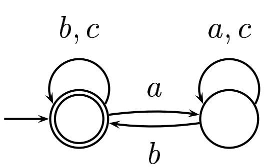


Run $aacb$ is accepting.

Automata on Finite Words

Finite automaton $\mathcal{A}$ over finite words is a tuple $(\Sigma, Q, \Delta, Q^0, F)$ where

- Σ is a finite alphabet
- Q is a finite set of states
- $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation
- $Q^0 \subseteq Q$ is a set of initial states
- $F \subseteq Q$ is a set of final states



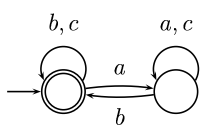
$\Sigma = \{a, b, c\}$, $Q = \{q_0, q_1\}$, $Q^0 = \{q_0\}$, $F = \{q_1\}$.

Automata on Infinite Words

- Reactive programs execute forever – so we want infinite sequences of states.
- Answer: finite automata over infinite words.
- Simplest case: Buchi automata
  - Same structure as automata on finite words
    - ... but different notion of acceptance
  - Recognize words from Σ^ω
    - $\Sigma = \{a, b\} \quad v = abaabaaab\ldots$
    - $\Sigma = \{a, b, c\} \quad \mathcal{L}_1 \subseteq \Sigma^\omega$ is $v \in \mathcal{L}_1$ iff after any occurrence of letter a there is some occurrence of letter b in v.
      Possible strings:
      $ababab\ldots \qquad aaabaaab\ldots$
      $abbabbabb\ldots \qquad accbaccb\ldots$

Automata on Finite Words, Cont'd

The language $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ is all words accepted by $\mathcal{A}$.

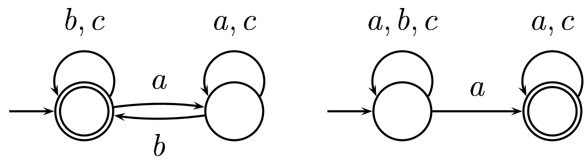


$\epsilon + a(a+c)^*b(b+c)^*$. This is a regular expression.

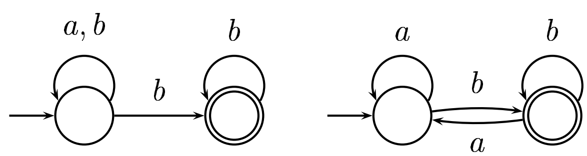
- Languages represented by regular expressions (and recognizable by finite automata on finite strings) are *regular* languages.
- An automaton is *deterministic* if $\forall a \cdot (q, a, q') \in \Delta \ \wedge \ (q, a, q'') \in \Delta \Rightarrow q' = q''$.
- Otherwise, it is *non-deterministic*.
- Every non-deterministic automaton on finite words can be translated into an equivalent deterministic automaton (which accepts the same language).

Operations on Buchi Automata

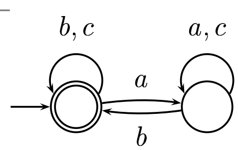
- Buchi-recognizable languages are closed under complementation.
    - i.e., from a Buchi automaton $\mathcal{A}$ recognizing $\mathcal{L}$ one can construct an automaton recognizing $\Sigma^\omega - \mathcal{L}$.
    - The number of states in this automaton is $O(2^{QlogQ})$, where Q – states in $\mathcal{A}$ (Safra's construction)
- Easy to do this for deterministic Buchi automata:



- Unfortunately, not all non-deterministic Buchi automata can be made deterministic!



Automata on Infinite Words (Cont'd)



Accepting language: $((b+c)^\omega a(a+c)^* b)^\omega$ (ω-regular expression)

- F – the set of *accepting* states
- A run of a Buchi automaton $\mathcal{A}$ over an infinite word $v \in \Sigma^\omega$. Domain of run – the set of all natural numbers.
- *inf*(ρ) – set of states that appear infinitely often in the run ρ. A run ρ is *accepting* (Buchi accepting) iff *inf*$(\rho) \cap F \neq \emptyset$.
- Language expressible by ω-regular expressions (and thus recognizable by some Buchi automaton) is ω-regular or Buchi-recognizable.

Operations on Buchi Automata, Cont'd

Buchi automata are closed under intersection [Chouka74]:

- given two Buchi automata $\mathcal{B}_1 = (\Sigma, Q_1, \Delta_1, Q_1^0, Q_1)$ (all states are accepting) and $\mathcal{B}_2 = (\Sigma, Q_2, \Delta_2, Q_2^0, F_2)$, construct $\mathcal{B}_1 \cap \mathcal{B}_2 = (\Sigma, Q_1 \times Q_2, \Delta', Q_1^0 \times Q_2^0, Q_1 \times F_2)$, where
  - $((r_i, q_j), a, (r_m, q_n)) \in \Delta'$ iff $(r_i, a, r_m) \in \Delta_1$ and $(q_j, a, q_n) \in \Delta_2$.

lementation Algorithm for Deterministic Au

Create two copies of an automaton:

- A_1: Take non-accepting states of A and make them accepting.
- A_2: Every transition to non-accepting state gets duplicated to same state in A_1.

Operations, Cont'd

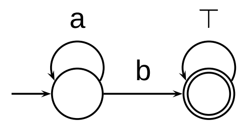
- The emptiness problem for Buchi automata is decidable
  - $\mathcal{L}(\mathcal{A}) \neq \emptyset$
  - logspace-complete for NLOGSPACE, i.e., solvable in linear time [Vardi, Wolper]) – see later in the lecture.
- Nonuniversality problem for Buchi automata is decidable
  - $\mathcal{L}(\mathcal{A}) \neq \Sigma^\omega$
  - logspace-complete for PSPACE [Sisla, Vardi, Wolper]

Intersection of arbitrary Buchi automata

- Main point: determining accepting states: need to go through accepting states of $\mathcal{B}_1$ and $\mathcal{B}_2$ infinite number of times
- 3 copies of the automaton:
  - 1st copy: start and accept here
  - 2nd copy: move when accepting state from B_1 has been seen
  - 3rd copy: move when accepting state from B_2 has been seen

LTL and Buchi Automata

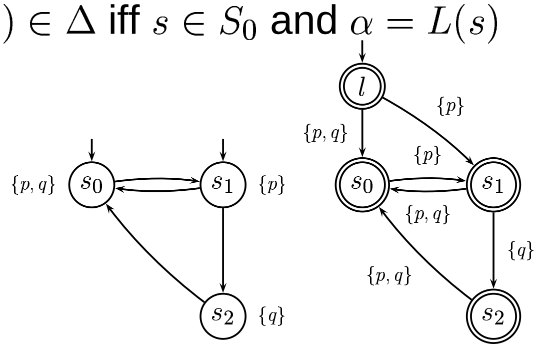
- Specification – also in the form of an automaton!
- Buchi automata can encode all LTL properties.
- Examples:



  - $a\ U\ b$
- Other examples:
  - $\Box \diamond p$
  - $\Box \diamond (p \vee q)$
  - $\neg \Box \diamond (p \vee q)$
  - $\neg(\Box(p\ U\ q))$

Modeling Systems Using Automata

- A system is a set of all its executions. So, every state is accepting!
- Transform Kripke structure (S, R, S_0, L)
  - where $L : S \to s^{AP}$
- ...into automaton $\mathcal{A} = (\Sigma, S \cup \{\ell\}, \Delta, \{\ell\}, S \cup \{\ell\})$,
  - where $\Sigma = 2^{AP}$
  - $(s, \alpha, s) \in \Delta$ for $s, s' \in S$ iff $(s, s') \in R$ and $\alpha = L(s')$
  - $(\ell, \alpha, s') \in \Delta$ iff $s \in S_0$ and $\alpha = L(s)$



Kripke structure Automaton

Sketch of the Algorithm

- Compute the set of subformulas that must hold in each reachable state and in each of its successor states.
  - Convert formula into normal form (negation for atomic propositions)
  - Create initial state, marked with the formula to be matched and a dummy incoming edge
  - Recursively
    - take a subformula that remains to be satisfied
    - look at the leading temporal operator: may split the current state into two, each annotated with appropriate subformula
  - Make connections to accepting state

LTL to Buchi Automata

- Theorem [Wolper, Vardi, Sisla 83]: Given an LTL formula ϕ, one can build a Buchi automaton $\mathcal{S} = (\Sigma, Q, \Delta, Q_0, F)$ where
  - $\Sigma = 2^{\text{Prop}}$
    - the number of automatic propositions, variables, etc. in ϕ
  - $|Q| \leq 2^{O(|\phi|)}$
    - $|\phi|$ - length of the formula
- ... s.t. $\mathcal{L}(\mathcal{S})$ is exactly the set of computations satisfying the formula ϕ.
- Algorithm given in Section 9.4
- But Buchi automata are more expressive than LTL!

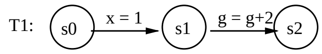
Complexity

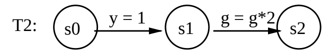
- Checking whether a formula ϕ is satisfied by a finite-state model K can be done in time $O(||K|| \times 2^{O(|\phi|)})$ or in space $O((log||K|| + \|\phi\|)^2)$.
- i.e., checking is polynomial in the size of the model and exponential in the size of the specification.

Automata-theoretic Model Checking

- The system $\mathcal{A}$ satisfies the specification $\mathcal{S}$ when
  - $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$
  - ... each behavior of the system is among the allowed behaviours
- Alternatively,
  - let $\overline{\mathcal{L}(\mathcal{S})}$ be the language $\Sigma^\omega - \mathcal{L}(\mathcal{S})$. Then, we are looking for
    - $\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$
    - no behavior of $\mathcal{A}$ is disallowed by $\mathcal{S}$.
  - If the intersection is not empty, any behavior in it corresponds to a counterexample.
  - Counterexample is always of the form uv^ω, where u and v are finite words.

Partial-order Reduction

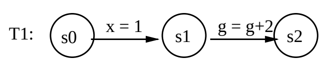
Example: 

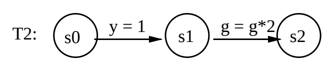


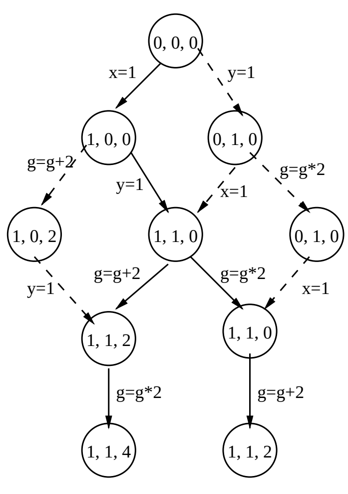
Dependent operations	Independent operations
g=g*2, g=g+2 (same data object)	x=1, y=1
x=1, g=g+2 (part of T1)	x=1, g=g*2
y=1, g=g*2 (part of T2)	y=1, g=g+2

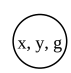
- 1 and 2 – differ only in relative order of y=1 and g=g+2 which are independent
- 4 and 5 – only relative order of x=1, g=g+2 which are independent
- Only 2 distinct runs:
  - 2. x = 1, y = 1, g = g+2, g = g*2
  - 3. x = 1, y = 1, g = g*2, g = g+2

Partial-order Reduction

Example: 







Interleaving:
Run sequences:

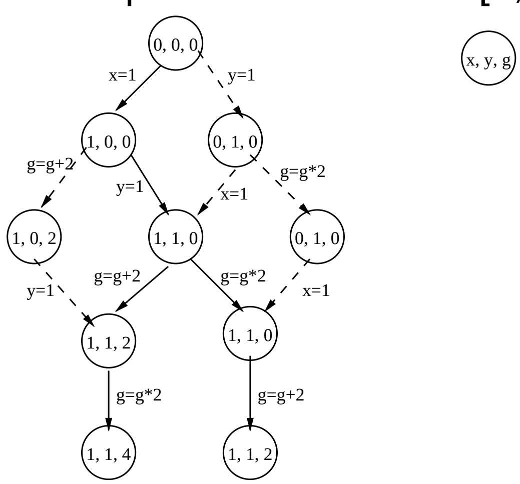
1. x=1, g=g+2, y=1, g=g*2
2. x=1, y=1, g=g+2, g=g*2
3. x=1, y=1, g=g*2, g=g+2
4. y=1, g=g*2, x=1, g=g*2
5. y=1, x=1, g=g*2, g=g+2
6. y=1, x=1, g=g+2, g=g*2

Closure Under Stuttering

- *Stuttering* refers to a sequence of identically labeled states along a path in a Kripke structure.
- Intuitively, an LTL formula is *closed under stuttering* if the interpretation of the formula remains the same under state sequences that differ only by repeated states [Abadi,Lamport'01].
  - Assume F is closed under stuttering. Then,
    - $\Box F$ is closed under stuttering
    - $\circ F$ is not closed under stuttering

Partial-order Reduction

- Two equivalence classes: [1, 2, 6], [3, 4, 5]



- For verification, it is sufficient to consider just one run from each equivalence class...
  - as long as the formulas are closed under stuttering!

f

ood for slide eater

Closure under stuttering and LTL$_{-X}$

LTL$_{-X}$ – a subset of LTL without the $\circ$ operator.

- Theorem: All LTL$_{-X}$ formulas are closed under stuttering [Lamport'94]
- Theorem: All cus LTL properties can be expressed in LTL$_{-X}$
  - By exponentially increasing the size of the formula!
- Determining whether an arbitrary LTL formula is closed under stuttering is PSPACE-complete [Peled, Wilke, Wolper'96]
- Exists an algorithm based on *edges* (changes in values of variables) that allows to use full LTL and yet guarantee closure under stuttering [Paun,Chechik'01]
  - Observation: stuttering does not add or delete edges or change their relative order
  - Theorem [Paun99]: If A and B are cus then so is $\diamond(\neg A \ \wedge \ \circ A \ \wedge \ \circ B)$