Formal Methods
Module I: Automated Reasoning
Ch. 04: **Automata-Theoretic LTL Reasoning**

Roberto Sebastiani and Stefano Tonetta

DISI, Università di Trento, Italy – `roberto.sebastiani@unitn.it`
URL: `https://disi.unitn.it/rseba/DIDATTICA/fm2024/`
Teaching assistant: Giuseppe Spallitta – `giuseppe.spallitta@unitn.it`

M.S. in Computer Science, Mathematics, & Artificial Intelligence Systems
Academic year 2023-2024

last update: Tuesday 16th April, 2024, 09:45

Copyright notice: *some material (text, figures) displayed in these slides is courtesy of R. Alur, M. Benerecetti, A. Cimatti, M. Di Natale, P. Pandya, M. Pistore, M. Roveri, C. Tinelli, and S.Tonetta, who detain its copyright. Some exampes displayed in these slides are taken from [Clarke, Grunberg & Peled, "Model Checking", MIT Press], and their copyright is detained by the authors. All the other material is copyrighted by Roberto Sebastiani. Every commercial use of this material is strictly forbidden by the copyright laws without the authorization of the authors. No copy of these slides can be displayed in public without containing this copyright notice.*

Outline

1. Büchi Automata

2. The Automata-Theoretic Approach to LTL Reasoning
   - General Ideas
   - Language-Emptiness Checking of Büchi Automata
   - From Kripke Models to Büchi Automata
   - From LTL Formulas to Büchi Automata
   - Complexity

3. Exercises

Outline

1. Büchi Automata

2. The Automata-Theoretic Approach to LTL Reasoning
   - General Ideas
   - Language-Emptiness Checking of Büchi Automata
   - From Kripke Models to Büchi Automata
   - From LTL Formulas to Büchi Automata
   - Complexity

3. Exercises

Infinite Word Languages

Modeling infinite computations of reactive systems

Given an Alphabet Σ (e.g. $\Sigma \overset{\text{def}}{=} \{a, b\}$)

- An ω-word α over Σ is an infinite sequence

  $$a_0,\ a_1,\ a_2 \ldots.$$

  Formally, $\alpha : \mathbb{N} \to \Sigma$.
- The set of all infinite words is denoted by Σ^ω.
- A ω-language L is collection of ω-words, i.e. $L \subseteq \Sigma^\omega$.
- Example: All words over $\{a, b\}$ with infinitely many a's.

Notation:

omega words $\alpha, \beta, \gamma \in \Sigma^\omega$.

omega-languages $L, L_1 \subseteq \Sigma^\omega$

For $u \in \Sigma^+$, let $u^\omega = u.u.u\ldots$

Infinite Word Languages

Modeling infinite computations of reactive systems

Given an Alphabet Σ (e.g. $\Sigma \overset{\text{def}}{=} \{a, b\}$)

- An ω-word α over Σ is an infinite sequence

  $a_0,\ a_1,\ a_2 \ldots.$

  Formally, $\alpha : \mathbb{N} \rightarrow \Sigma$.
- The set of all infinite words is denoted by Σ^ω.
- A ω-language L is collection of ω-words, i.e. $L \subseteq \Sigma^\omega$.
- Example: All words over $\{a, b\}$ with infinitely many a's.

Notation:

omega words $\alpha, \beta, \gamma \in \Sigma^\omega$.

omega-languages $L, L_1 \subseteq \Sigma^\omega$

For $u \in \Sigma^+$, let $u^\omega = u.u.u \ldots$

Infinite Word Languages

Modeling infinite computations of reactive systems

Given an Alphabet Σ (e.g. $\Sigma \overset{\text{def}}{=} \{a, b\}$)

- An ω-word α over Σ is an infinite sequence

  $a_0,\ a_1,\ a_2 \ldots.$

  Formally, $\alpha : \mathbb{N} \to \Sigma$.
- The set of all infinite words is denoted by Σ^ω.
- A ω-language L is collection of ω-words, i.e. $L \subseteq \Sigma^\omega$.
- Example: All words over $\{a, b\}$ with infinitely many a's.

Notation:

omega words $\alpha, \beta, \gamma \in \Sigma^\omega$.

omega-languages $L, L_1 \subseteq \Sigma^\omega$

For $u \in \Sigma^+$, let $u^\omega = u.u.u \ldots$

Infinite Word Languages

Modeling infinite computations of reactive systems

Given an Alphabet Σ (e.g. $\Sigma \overset{\text{def}}{=} \{a, b\}$)

- An ω-word α over Σ is an infinite sequence

  $$a_0,\ a_1,\ a_2 \ldots.$$

  Formally, $\alpha : \mathbb{N} \rightarrow \Sigma$.
- The set of all infinite words is denoted by Σ^ω.
- A ω-language L is collection of ω-words, i.e. $L \subseteq \Sigma^\omega$.
- Example: All words over $\{a, b\}$ with infinitely many a's.

Notation:

omega words $\alpha, \beta, \gamma \in \Sigma^\omega$.

omega-languages $L, L_1 \subseteq \Sigma^\omega$

For $u \in \Sigma^+$, let $u^\omega = u.u.u\ldots$

Infinite Word Languages

Modeling infinite computations of reactive systems

Given an Alphabet Σ (e.g. $\Sigma \overset{\text{def}}{=} \{a, b\}$)

- An ω-word α over Σ is an infinite sequence

  $$a_0,\ a_1,\ a_2 \ldots.$$

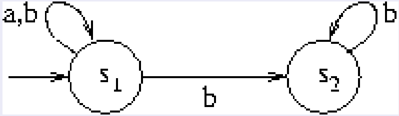
  Formally, $\alpha : \mathbb{N} \rightarrow \Sigma$.
- The set of all infinite words is denoted by Σ^ω.
- A ω-language L is collection of ω-words, i.e. $L \subseteq \Sigma^\omega$.
- Example: All words over $\{a, b\}$ with infinitely many a's.

Notation:

omega words $\alpha, \beta, \gamma \in \Sigma^\omega$.
omega-languages $L, L_1 \subseteq \Sigma^\omega$
For $u \in \Sigma^+$, let $u^\omega = u.u.u \ldots$

Omega-Automata

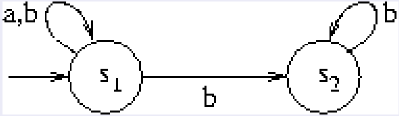
- We consider automaton running over infinite words.



- Let $\alpha = aabbbb\ldots$.
  There are several (infinite) possible runs.
  Run $\rho_1 = s_1, s_1, s_1, s_1, s_2, s_2 \ldots$
  Run $\rho_2 = s_1, s_1, s_1, s_1, s_1, s_1 \ldots$
- Acceptance Conditions: Büchi (Muller, Rabin, Street):
  Acceptance is based on states occurring infinitely often
- Notation: Let Q be the set of states. Let $\rho \in Q^\omega$. Then,
  $$Inf(\rho) = \{s \in Q \mid \exists^\infty i \in \mathbb{N}.\ \rho(i) = s\}.$$
  (The set of states occurring infinitely many times in ρ.)

Omega-Automata

- We consider automaton running over infinite words.



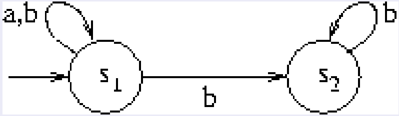
- Let $\alpha \; = \; aabbbb\ldots$.
  There are several (infinite) possible runs.
  Run $\rho_1 \; = \; s_1, s_1, s_1, s_1, s_2, s_2 \ldots$
  Run $\rho_2 \; = \; s_1, s_1, s_1, s_1, s_1, s_1 \ldots$
- Acceptance Conditions: Büchi (Muller, Rabin, Street):
  Acceptance is based on states occurring infinitely often
- Notation: Let Q be the set of states. Let $\rho \in Q^\omega$. Then,

$$Inf(\rho) \; = \; \{s \in Q \mid \exists^\infty i \in \mathbb{N}.\; \rho(i) = s\}.$$

  (The set of states occurring infinitely many times in ρ.)

Omega-Automata

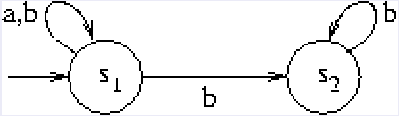
- We consider automaton running over infinite words.



- Let $\alpha = aabbbb\ldots$.
  There are several (infinite) possible runs.
  Run $\rho_1 = s_1, s_1, s_1, s_1, s_2, s_2 \ldots$
  Run $\rho_2 = s_1, s_1, s_1, s_1, s_1, s_1 \ldots$
- Acceptance Conditions: Büchi (Muller, Rabin, Street):
  Acceptance is based on states occurring infinitely often
- Notation: Let Q be the set of states. Let $\rho \in Q^\omega$. Then,
  $$Inf(\rho) = \{s \in Q \mid \exists^\infty i \in \mathbb{N}.\ \rho(i) = s\}.$$
  (The set of states occurring infinitely many times in ρ.)

Omega-Automata

- We consider automaton running over infinite words.



- Let $\alpha = aabbbb\ldots$.
  There are several (infinite) possible runs.
  Run $\rho_1 = s_1, s_1, s_1, s_1, s_2, s_2 \ldots$
  Run $\rho_2 = s_1, s_1, s_1, s_1, s_1, s_1 \ldots$
- Acceptance Conditions: Büchi (Muller, Rabin, Street):
  Acceptance is based on states occurring infinitely often
- Notation: Let Q be the set of states. Let $\rho \in Q^\omega$. Then,
  $$Inf(\rho) = \{s \in Q \mid \exists^\infty i \in \mathbb{N}.\ \rho(i) = s\}.$$
  (The set of states occurring infinitely many times in ρ.)

Büchi Automata

Nondeterministic Büchi Automaton

- A **Nondeterministic Büchi Automaton (NBA)** is $(Q, \Sigma, \delta, I, F)$ s.t.
  - Q Finite set of states.
  - Σ is a finite alphabet
  - $I \subseteq Q$ set of initial states.
  - $F \subseteq Q$ set of accepting states.
  - $\delta \subseteq Q \times \Sigma \times Q$ transition relation (edges).
- A **Deterministic Büchi Automaton (DBA)** is an NBA s.t. the transition relation is functional: $\delta : Q \times \Sigma \longmapsto Q$

Runs and Language of NBAs

- A run ρ of A on ω-word $\alpha = a_0, a_1, a_2, \ldots$ is an infinite sequence $\rho = q_0, q_1, q_2, \ldots$ s.t. $q_0 \in I$ and $q_i \xrightarrow{a_i} q_{i+1}$ for $0 \leq i$.
- The run ρ is accepting if

  $$Inf(\rho) \cap F \neq \emptyset.$$

- The language accepted by A $\mathcal{L}(A) = \{\alpha \in \Sigma^\omega \mid A \text{ has an accepting run on } \alpha\}$

Büchi Automata

Nondeterministic Büchi Automaton

- A Nondeterministic Büchi Automaton (NBA) is $(Q, \Sigma, \delta, I, F)$ s.t.
  - Q Finite set of states.
  - Σ is a finite alphabet
  - $I \subseteq Q$ set of initial states.
  - $F \subseteq Q$ set of accepting states.
  - $\delta \subseteq Q \times \Sigma \times Q$ transition relation (edges).
- A Deterministic Büchi Automaton (DBA) is an NBA s.t. the transition relation is functional: $\delta : Q \times \Sigma \longmapsto Q$

Runs and Language of NBAs

- A run ρ of A on ω-word $\alpha = a_0, a_1, a_2, \ldots$ is an infinite sequence $\rho = q_0, q_1, q_2, \ldots$ s.t. $q_0 \in I$ and $q_i \xrightarrow{a_i} q_{i+1}$ for $0 \leq i$.
- The run ρ is accepting if
  $$Inf(\rho) \cap F \neq \emptyset.$$
- The language accepted by A $\mathcal{L}(A) = \{\alpha \in \Sigma^\omega \mid A \text{ has an accepting run on } \alpha\}$

Büchi Automata

Nondeterministic Büchi Automaton

- A Nondeterministic Büchi Automaton (NBA) is $(Q, \Sigma, \delta, I, F)$ s.t.
  - Q Finite set of states.
  - Σ is a finite alphabet
  - $I \subseteq Q$ set of initial states.
  - $F \subseteq Q$ set of accepting states.
  - $\delta \subseteq Q \times \Sigma \times Q$ transition relation (edges).
- A Deterministic Büchi Automaton (DBA) is an NBA s.t. the transition relation is functional: $\delta : Q \times \Sigma \longmapsto Q$

Runs and Language of NBAs

- A run ρ of A on ω-word $\alpha = a_0, a_1, a_2, \ldots$ is an infinite sequence $\rho = q_o, q_1, q_2, \ldots$ s.t. $q_0 \in I$ and $q_i \xrightarrow{a_i} q_{i+1}$ for $0 \leq i$.
- The run ρ is accepting if

$$Inf(\rho) \cap F \neq \emptyset.$$

- The language accepted by A $\mathcal{L}(A) = \{\alpha \in \Sigma^\omega \mid A \text{ has an accepting run on } \alpha\}$

Büchi Automata

Nondeterministic Büchi Automaton

- A Nondeterministic Büchi Automaton (NBA) is $(Q, \Sigma, \delta, I, F)$ s.t.
  - Q Finite set of states.
  - Σ is a finite alphabet
  - $I \subseteq Q$ set of initial states.
  - $F \subseteq Q$ set of accepting states.
  - $\delta \subseteq Q \times \Sigma \times Q$ transition relation (edges).
- A Deterministic Büchi Automaton (DBA) is an NBA s.t. the transition relation is functional: $\delta : Q \times \Sigma \longmapsto Q$

Runs and Language of NBAs

- A run ρ of A on ω-word $\alpha = a_0, a_1, a_2, \ldots$ is an infinite sequence $\rho = q_o, q_1, q_2, \ldots$ s.t. $q_0 \in I$ and $q_i \xrightarrow{a_i} q_{i+1}$ for $0 \leq i$.
- The run ρ is accepting if

  $$Inf(\rho) \cap F \neq \emptyset.$$

- The language accepted by A
  $\mathcal{L}(A) = \{\alpha \in \Sigma^\omega \mid A \text{ has an accepting run on } \alpha\}$

Büchi Automata

Nondeterministic Büchi Automaton

- A Nondeterministic Büchi Automaton (NBA) is $(Q, \Sigma, \delta, I, F)$ s.t.
  - Q Finite set of states.
  - Σ is a finite alphabet
  - $I \subseteq Q$ set of initial states.
  - $F \subseteq Q$ set of accepting states.
  - $\delta \subseteq Q \times \Sigma \times Q$ transition relation (edges).
- A Deterministic Büchi Automaton (DBA) is an NBA s.t. the transition relation is functional: $\delta : Q \times \Sigma \longmapsto Q$

Runs and Language of NBAs

- A run ρ of A on ω-word $\alpha = a_0, a_1, a_2, \ldots$ is an infinite sequence $\rho = q_o, q_1, q_2, \ldots$ s.t. $q_0 \in I$ and $q_i \xrightarrow{a_i} q_{i+1}$ for $0 \leq i$.
- The run ρ is accepting if

$$Inf(\rho) \cap F \neq \emptyset.$$

- The language accepted by A $\mathcal{L}(A) = \{\alpha \in \Sigma^\omega \mid A \text{ has an accepting run on } \alpha\}$

Büchi Automata

Nondeterministic Büchi Automaton

- A Nondeterministic Büchi Automaton (NBA) is $(Q, \Sigma, \delta, I, F)$ s.t.
  - Q Finite set of states.
  - Σ is a finite alphabet
  - $I \subseteq Q$ set of initial states.
  - $F \subseteq Q$ set of accepting states.
  - $\delta \subseteq Q \times \Sigma \times Q$ transition relation (edges).
- A Deterministic Büchi Automaton (DBA) is an NBA s.t. the transition relation is functional: $\delta : Q \times \Sigma \longmapsto Q$

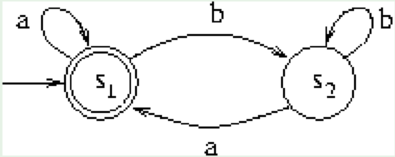
Runs and Language of NBAs

- A run ρ of A on ω-word $\alpha = a_0, a_1, a_2, ...$ is an infinite sequence $\rho = q_o, q_1, q_2, \ldots$ s.t. $q_0 \in I$ and $q_i \xrightarrow{a_i} q_{i+1}$ for $0 \leq i$.
- The run ρ is accepting if

  $$Inf(\rho) \cap F \neq \emptyset.$$
- The language accepted by A $\mathcal{L}(A) = \{\alpha \in \Sigma^\omega \mid A \text{ has an accepting run on } \alpha\}$

Büchi Automaton: Example

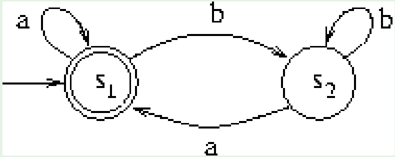
Let $\Sigma = \{a, b\}$.
Let a Deterministic Büchi Automaton (DBA) A_1 be



- With $F = \{s_1\}$ the automaton recognizes words with infinitely many a's.
- With $F = \{s_2\}$ the automaton recognizes words with infinitely many b's.

Büchi Automaton: Example

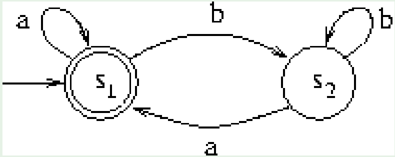
Let $\Sigma = \{a, b\}$.
Let a Deterministic Büchi Automaton (DBA) A_1 be



- With $F = \{s_1\}$ the automaton recognizes words with infinitely many a's.
- With $F = \{s_2\}$ the automaton recognizes words with infinitely many b's.

Büchi Automaton: Example

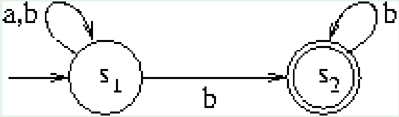
Let $\Sigma = \{a, b\}$.
Let a Deterministic Büchi Automaton (DBA) A_1 be



- With $F = \{s_1\}$ the automaton recognizes words with infinitely many a's.
- With $F = \{s_2\}$ the automaton recognizes words with infinitely many b's.

Büchi Automaton: Example (2)

Let a Nondeterministic Büchi Automaton (NBA) A_2 be



With $F = \{s_2\}$, the automaton A_2 recognizes words with finitely many a. Thus, $\mathcal{L}(A_2) = \overline{\mathcal{L}(A_1)}$.

Deterministic vs. Nondeterministic Büchi Automata

Theorem

*DBA*s are strictly less powerful than *NBA*s.

Remark:

The subset construction of standard Final-State automata does not work!

Let DA_2 be

- DA_2 is not equivalent to A_2 (e.g., it recognizes $(b.a)^\omega$)
- There is no DBA equivalent to A_2

Deterministic vs. Nondeterministic Büchi Automata

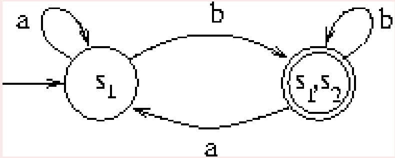
Theorem

*DBA*s are strictly less powerful than *NBA*s.

Remark:

The subset construction of standard Final-State automata does not work!

Let DA_2 be



- DA_2 is not equivalent to A_2
  (e.g., it recognizes $(b.a)^\omega$)
- There is no DBA equivalent to A_2

Deterministic vs. Nondeterministic Büchi Automata

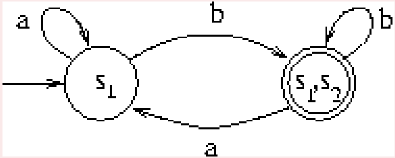
Theorem

*DBA*s are strictly less powerful than *NBA*s.

Remark:

The subset construction of standard Final-State automata does not work!

Let DA_2 be



- DA_2 is not equivalent to A_2 (e.g., it recognizes $(b.a)^\omega$)
- There is no DBA equivalent to A_2

Deterministic vs. Nondeterministic Büchi Automata

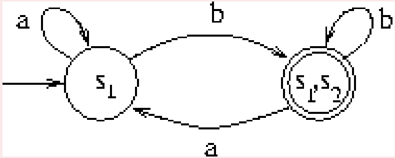
Theorem

*DBA*s are strictly less powerful than *NBA*s.

Remark:

The subset construction of standard Final-State automata does not work!

Let DA_2 be



- DA_2 is not equivalent to A_2 (e.g., it recognizes $(b.a)^\omega$)
- There is no DBA equivalent to A_2

Closure Properties

Theorem (union, intersection)

For the NBAs A_1, A_2 we can construct

- the NBA A s.t. $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. $|A| = |A_1| + |A_2|$
- the NBA A s.t. $\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. $|A| \leq |A_1| \cdot |A_2| \cdot 2$.

Closure Properties

Theorem (union, intersection)

For the NBAs A_1, A_2 we can construct

- the NBA A s.t. $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. $|A| = |A_1| + |A_2|$
- the NBA A s.t. $\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. $|A| \leq |A_1| \cdot |A_2| \cdot 2$.

Closure Properties

Theorem (union, intersection)

For the NBAs A_1, A_2 we can construct

- the NBA A s.t. $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. $|A| = |A_1| + |A_2|$
- the NBA A s.t. $\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. $|A| \leq |A_1| \cdot |A_2| \cdot 2$.

Union of two NBAs

Definition: union of NBAs

Let $A_1 = (Q_1, \Sigma_1, \delta_1, I_1, F_1)$, $A_2 = (Q_2, \Sigma_2, \delta_2, I_2, F_2)$.
Then $A = A_1 \cup A_2 = (Q, \Sigma, \delta, I, F)$ is defined as follows

- $Q := Q_1 \cup Q_2$, $I := I_1 \cup I_2$, $F := F_1 \cup F_2$
- $R(s, s') := \begin{cases} R_1(s, s') \text{ if } s \in Q_1 \\ R_2(s, s') \text{ if } s \in Q_2 \end{cases}$

Theorem

- $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$
- $|A| = |A_1| + |A_2|$

Note

A is an automaton which just runs nondeterministically either A_1 or A_2
(same construction as with ordinary automata)

Union of two NBAs

Definition: union of NBAs

Let $A_1 = (Q_1, \Sigma_1, \delta_1, I_1, F_1)$, $A_2 = (Q_2, \Sigma_2, \delta_2, I_2, F_2)$.
Then $A = A_1 \cup A_2 = (Q, \Sigma, \delta, I, F)$ is defined as follows

- $Q := Q_1 \cup Q_2$, $I := I_1 \cup I_2$, $F := F_1 \cup F_2$
- $R(s, s') := \begin{cases} R_1(s, s') \text{ if } s \in Q_1 \\ R_2(s, s') \text{ if } s \in Q_2 \end{cases}$

Theorem

- $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$
- $|A| = |A_1| + |A_2|$

Note

A is an automaton which just runs nondeterministically either A_1 or A_2
(same construction as with ordinary automata)

Union of two NBAs

Definition: union of NBAs

Let $A_1 = (Q_1, \Sigma_1, \delta_1, I_1, F_1)$, $A_2 = (Q_2, \Sigma_2, \delta_2, I_2, F_2)$.
Then $A = A_1 \cup A_2 = (Q, \Sigma, \delta, I, F)$ is defined as follows

- $Q := Q_1 \cup Q_2$, $I := I_1 \cup I_2$, $F := F_1 \cup F_2$
- $R(s, s') := \begin{cases} R_1(s, s') \text{ if } s \in Q_1 \\ R_2(s, s') \text{ if } s \in Q_2 \end{cases}$

Theorem

- $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$
- $|A| = |A_1| + |A_2|$

Note

A is an automaton which just runs nondeterministically either A_1 or A_2
(same construction as with ordinary automata)

Union of two NBAs

Definition: union of NBAs

Let $A_1 = (Q_1, \Sigma_1, \delta_1, I_1, F_1)$, $A_2 = (Q_2, \Sigma_2, \delta_2, I_2, F_2)$.
Then $A = A_1 \cup A_2 = (Q, \Sigma, \delta, I, F)$ is defined as follows

- $Q := Q_1 \cup Q_2$, $I := I_1 \cup I_2$, $F := F_1 \cup F_2$
- $R(s, s') := \begin{cases} R_1(s, s') \text{ if } s \in Q_1 \\ R_2(s, s') \text{ if } s \in Q_2 \end{cases}$

Theorem

- $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$
- $|A| = |A_1| + |A_2|$

Note

A is an automaton which just runs nondeterministically either A_1 or A_2
(same construction as with ordinary automata)

Union of two NBAs

Definition: union of NBAs

Let $A_1 = (Q_1, \Sigma_1, \delta_1, I_1, F_1)$, $A_2 = (Q_2, \Sigma_2, \delta_2, I_2, F_2)$.
Then $A = A_1 \cup A_2 = (Q, \Sigma, \delta, I, F)$ is defined as follows

- $Q := Q_1 \cup Q_2$, $I := I_1 \cup I_2$, $F := F_1 \cup F_2$
- $R(s, s') := \begin{cases} R_1(s, s') \text{ if } s \in Q_1 \\ R_2(s, s') \text{ if } s \in Q_2 \end{cases}$

Theorem

- $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$
- $|A| = |A_1| + |A_2|$

Note

A is an automaton which just runs nondeterministically either A_1 or A_2
(same construction as with ordinary automata)

Synchronous Product of NBAs

Definition: synchronous product of NBAs

Let $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$.
Then, $A_1 \times A_2 = (Q, \Sigma, \delta, I, F)$, where

$$Q = Q_1 \times Q_2 \times \{1, 2\}.$$
$$I = I_1 \times I_2 \times \{1\}.$$
$$F = F_1 \times Q_2 \times \{1\}.$$

$\langle p, q, 1\rangle \xrightarrow{a} \langle p', q', 1\rangle$ iff $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ and $p \notin F_1$.
$\langle p, q, 1\rangle \xrightarrow{a} \langle p', q', 2\rangle$ iff $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ and $p \in F_1$.
$\langle p, q, 2\rangle \xrightarrow{a} \langle p', q', 2\rangle$ iff $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ and $q \notin F_2$.
$\langle p, q, 2\rangle \xrightarrow{a} \langle p', q', 1\rangle$ iff $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ and $q \in F_2$.

Theorem

- $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.
- $|A_1 \times A_2| \leq 2 \cdot |A_1| \cdot |A_2|$.

Synchronous Product of NBAs

Definition: synchronous product of NBAs

Let $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$.
Then, $A_1 \times A_2 = (Q, \Sigma, \delta, I, F)$, where

$$Q = Q_1 \times Q_2 \times \{1, 2\}.$$
$$I = I_1 \times I_2 \times \{1\}.$$
$$F = F_1 \times Q_2 \times \{1\}.$$

$\langle p, q, 1\rangle \xrightarrow{a} \langle p', q', 1\rangle$ iff $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ and $p \notin F_1$.
$\langle p, q, 1\rangle \xrightarrow{a} \langle p', q', 2\rangle$ iff $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ and $p \in F_1$.
$\langle p, q, 2\rangle \xrightarrow{a} \langle p', q', 2\rangle$ iff $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ and $q \notin F_2$.
$\langle p, q, 2\rangle \xrightarrow{a} \langle p', q', 1\rangle$ iff $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ and $q \in F_2$.

Theorem

- $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.
- $|A_1 \times A_2| \leq 2 \cdot |A_1| \cdot |A_2|$.

Synchronous Product of NBAs: Intuition

- The automaton remembers two tracks, one for each source NBA, and it points to one of the two tracks
- As soon as it goes through an accepting state of the current track, it switches to the other track

$\Longrightarrow$ to visit infinitely often a state in F (i.e., F_1), it must visit infinitely often some state also in F_2

- Important subcase: If $F_2 = Q_2$, then

$$Q = Q_1 \times Q_2.$$
$$I = I_1 \times I_2.$$
$$F = F_1 \times Q_2.$$

Synchronous Product of NBAs: Intuition

- The automaton remembers two tracks, one for each source NBA, and it points to one of the two tracks
- As soon as it goes through an accepting state of the current track, it switches to the other track

$\Longrightarrow$ to visit infinitely often a state in F (i.e., F_1), it must visit infinitely often some state also in F_2

- Important subcase: If $F_2 = Q_2$, then

$$Q = Q_1 \times Q_2.$$
$$I = I_1 \times I_2.$$
$$F = F_1 \times Q_2.$$

Synchronous Product of NBAs: Intuition

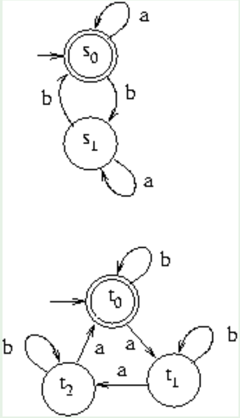
- The automaton remembers two tracks, one for each source NBA, and it points to one of the two tracks
- As soon as it goes through an accepting state of the current track, it switches to the other track

$\Longrightarrow$ to visit infinitely often a state in F (i.e., F_1), it must visit infinitely often some state also in F_2

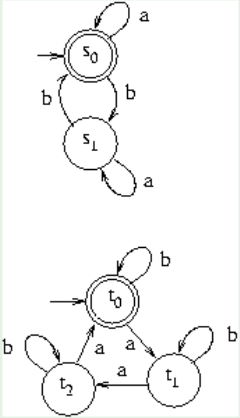
- Important subcase: If $F_2 = Q_2$, then

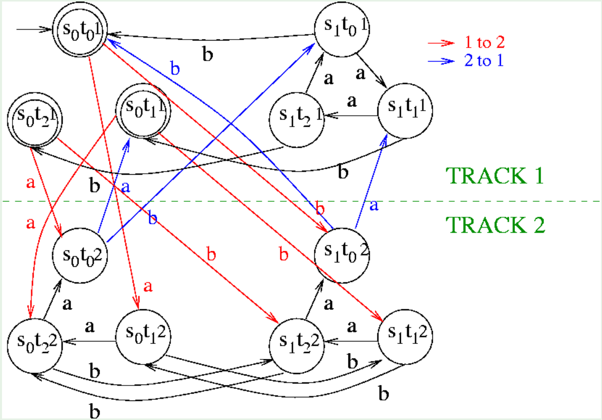
$$Q = Q_1 \times Q_2.$$
$$I = I_1 \times I_2.$$
$$F = F_1 \times Q_2.$$

Synchronous Product of NBAs: Example



Synchronous Product of NBAs: Example





Closure Properties (2)

Theorem (complementation) [Safra, MacNaughten]

For the NBA A_1 we can construct an NBA A_2 such that $\mathcal{L}(A_2) = \overline{\mathcal{L}(A_1)}$.
$|A_2| = O(2^{|A_1| \cdot \log(|A_1|)})$.

Method: (hint)

(i) convert a Büchi automaton into a Non-Deterministic Rabin automaton
(ii) determinize and Complement the Rabin automaton
(iii) convert the Rabin automaton into a Büchi automaton.

Closure Properties (2)

Theorem (complementation) [Safra, MacNaughten]

For the NBA A_1 we can construct an NBA A_2 such that $\mathcal{L}(A_2) = \overline{\mathcal{L}(A_1)}$.
$|A_2| = O(2^{|A_1| \cdot \log(|A_1|)})$.

Method: (hint)

(i) convert a Büchi automaton into a Non-Deterministic Rabin automaton
(ii) determinize and Complement the Rabin automaton
(iii) convert the Rabin automaton into a Büchi automaton.

Closure Properties (2)

Theorem (complementation) [Safra, MacNaughten]

For the NBA A_1 we can construct an NBA A_2 such that $\mathcal{L}(A_2) = \overline{\mathcal{L}(A_1)}$.
$|A_2| = O(2^{|A_1| \cdot \log(|A_1|)})$.

Method: (hint)

(i) convert a Büchi automaton into a Non-Deterministic Rabin automaton
(ii) determinize and Complement the Rabin automaton
(iii) convert the Rabin automaton into a Büchi automaton.

Closure Properties (2)

Theorem (complementation) [Safra, MacNaughten]

For the NBA A_1 we can construct an NBA A_2 such that $\mathcal{L}(A_2) = \overline{\mathcal{L}(A_1)}$.
$|A_2| = O(2^{|A_1| \cdot \log(|A_1|)})$.

Method: (hint)

(i) convert a Büchi automaton into a Non-Deterministic Rabin automaton
(ii) determinize and Complement the Rabin automaton
(iii) convert the Rabin automaton into a Büchi automaton.

Closure Properties (2)

Theorem (complementation) [Safra, MacNaughten]

For the NBA A_1 we can construct an NBA A_2 such that $\mathcal{L}(A_2) = \overline{\mathcal{L}(A_1)}$.
$|A_2| = O(2^{|A_1| \cdot \log(|A_1|)})$.

Method: (hint)

(i) convert a Büchi automaton into a Non-Deterministic Rabin automaton
(ii) determinize and Complement the Rabin automaton
(iii) convert the Rabin automaton into a Büchi automaton.

Generalized Büchi Automaton

Definition

- A Generalized Büchi Automaton is a tuple $A := (Q, \Sigma, \delta, I, FT)$ where $FT = \langle F_1, F_2, \ldots, F_k \rangle$ with $F_i \subseteq Q$.
- A run ρ of A is accepting if $Inf(\rho) \cap F_i \neq \emptyset$ for each $1 \leq i \leq k$.

Theorem

For every Generalized Büchi Automaton we can construct a language equivalent plain Büchi Automaton.

Intuition

Let $Q' = Q \times \{1, \ldots, K\}$.
The automaton remains in phase i till it visits a state in F_i. Then, it moves to $(i \ mod \ K) + 1$ mode.

Generalized Büchi Automaton

Definition

- A Generalized Büchi Automaton is a tuple $A := (Q, \Sigma, \delta, I, FT)$ where $FT = \langle F_1, F_2, \ldots, F_k \rangle$ with $F_i \subseteq Q$.
- A run ρ of A is accepting if $Inf(\rho) \cap F_i \neq \emptyset$ for each $1 \leq i \leq k$.

Theorem

For every Generalized Büchi Automaton we can construct a language equivalent plain Büchi Automaton.

Intuition

Let $Q' = Q \times \{1, \ldots, K\}$.
The automaton remains in phase i till it visits a state in F_i. Then, it moves to $(i \ mod \ K) + 1$ mode.

Generalized Büchi Automaton

Definition

- A Generalized Büchi Automaton is a tuple $A := (Q, \Sigma, \delta, I, FT)$ where $FT = \langle F_1, F_2, \ldots, F_k \rangle$ with $F_i \subseteq Q$.
- A run ρ of A is accepting if $Inf(\rho) \cap F_i \neq \emptyset$ for each $1 \leq i \leq k$.

Theorem

For every Generalized Büchi Automaton we can construct a language equivalent plain Büchi Automaton.

Intuition

Let $Q' = Q \times \{1, \ldots, K\}$.
The automaton remains in phase i till it visits a state in F_i. Then, it moves to $(i \; mod \; K) + 1$ mode.

De-generalization of a generalized NBA

Definition: De-generalization of a generalized NBA

Let $A \overset{\text{def}}{=} (Q, \Sigma, \delta, I, FT)$ a generalized BA s.f. $FT \overset{\text{def}}{=} \{F_1, ..., F_K\}$.
Then a language-equivalent BA $A' \overset{\text{def}}{=} (Q', \Sigma, \delta', I', F')$ is built as follows

$$\begin{array}{lcl} Q' & = & Q_1 \times \{1, ..., K\}. \\ I' & = & I \times \{1\}. \\ F' & = & F_1 \times \{1\}. \end{array}$$

δ' is s.t., for every $i \in [1, ..., K]$:

$$\begin{array}{llllll} \langle p, i\rangle \xrightarrow{a} \langle q, i\rangle & \text{iff} & p \xrightarrow{a} q & \in \delta & \text{and} & p \notin F_i. \\ \langle p, i\rangle \xrightarrow{a} \langle q, (i \; mod \; K) + 1\rangle & \text{iff} & p \xrightarrow{a} q & \in \delta & \text{and} & p \in F_i. \end{array}$$

Theorem

- $\mathcal{L}(A') = \mathcal{L}(A)$.
- $|A'| \leq K \cdot |A|$.

De-generalization of a generalized NBA

Definition: De-generalization of a generalized NBA

Let $A \overset{\text{def}}{=} (Q, \Sigma, \delta, I, FT)$ a generalized BA s.f. $FT \overset{\text{def}}{=} \{F_1, ..., F_K\}$.
Then a language-equivalent BA $A' \overset{\text{def}}{=} (Q', \Sigma, \delta', I', F')$ is built as follows

$$Q' \;=\; Q_1 \times \{1, ..., K\}.$$
$$I' \;=\; I \times \{1\}.$$
$$F' \;=\; F_1 \times \{1\}.$$

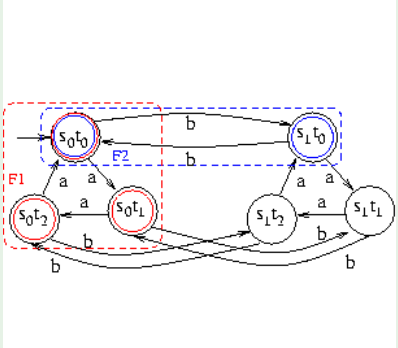
δ' is s.t., for every $i \in [1, ..., K]$:

$$\begin{array}{llllll} \langle p, i\rangle \xrightarrow{a} \langle q, i\rangle & \text{iff} & p \xrightarrow{a} q \in \delta & \text{and} & p \notin F_i. \\ \langle p, i\rangle \xrightarrow{a} \langle q, (i \; mod \; K) + 1\rangle & \text{iff} & p \xrightarrow{a} q \in \delta & \text{and} & p \in F_i. \end{array}$$

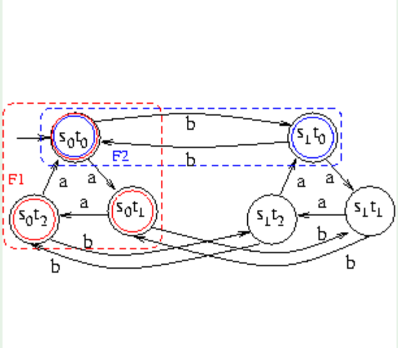
Theorem

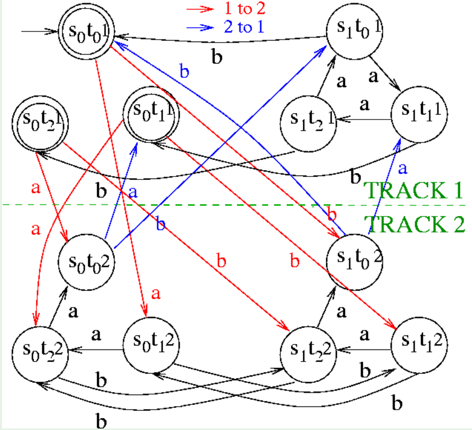
- $\mathcal{L}(A') \;=\; \mathcal{L}(A)$.
- $|A'| \leq K \cdot |A|$.

Degeneralizing a Büchi automaton: Example



Degeneralizing a Büchi automaton: Example





Omega-regular Expressions

Recall:

A finite-word language is called regular if it it is recognizable by some Finite-State-Automaton (FSA).

Definition

An infinite-word language is called ω-regular if it has the form $\cup_{i=1}^{n} \; U_i.(V_i)^\omega$ where U_i, V_i are regular languages.

Theorem

A language L is ω-regular iff it is NBA-recognizable.

Omega-regular Expressions

Recall:

A finite-word language is called regular if it it is recognizable by some Finite-State-Automaton (FSA).

Definition

An infinite-word language is called ω-regular if it has the form $\cup_{i=1}^{n} \; U_i.(V_i)^{\omega}$ where U_i, V_i are regular languages.

Theorem

A language L is ω-regular iff it is NBA-recognizable.

Outline

1. Büchi Automata

2. The Automata-Theoretic Approach to LTL Reasoning
   - General Ideas
   - Language-Emptiness Checking of Büchi Automata
   - From Kripke Models to Büchi Automata
   - From LTL Formulas to Büchi Automata
   - Complexity

3. Exercises

Outline

1. Büchi Automata

2. The Automata-Theoretic Approach to LTL Reasoning
    - General Ideas
    - Language-Emptiness Checking of Büchi Automata
    - From Kripke Models to Büchi Automata
    - From LTL Formulas to Büchi Automata
    - Complexity

3. Exercises

Automata-Theoretic LTL Satisfiability and Entailment

LTL Validity/Satisfiability

- Let ψ be an LTL formula

$$\begin{aligned} & \models \psi \quad \text{(LTL)} \\ \Longleftrightarrow\ & \neg\psi \text{ unsat} \\ \Longleftrightarrow\ & \mathcal{L}(A_{\neg\psi}) = \emptyset \end{aligned}$$

- $A_{\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\neg\psi$ (do not satisfy ψ)

LTL Entailment

- Let φ, ψ be an LTL formula

(LTL)

(LTL)

$\Longleftrightarrow$

$\Longleftrightarrow$

- $A_{\varphi\wedge\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\varphi \wedge \neg\psi$ (satisfy φ and do not satisfy ψ)

Automata-Theoretic LTL Satisfiability and Entailment

LTL Validity/Satisfiability

- Let ψ be an LTL formula

$$\models \psi \quad \text{(LTL)}$$
$$\Longleftrightarrow \neg\psi \text{ unsat}$$
$$\Longleftrightarrow \mathcal{L}(A_{\neg\psi}) = \emptyset$$

- $A_{\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\neg\psi$ (do not satisfy ψ)

LTL Entailment

- Let φ, ψ be an LTL formula

(LTL)

(LTL)

$\Longleftrightarrow$

$\Longleftrightarrow$

- $A_{\varphi\wedge\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\varphi \wedge \neg\psi$ (satisfy φ and do not satisfy ψ)

Automata-Theoretic LTL Satisfiability and Entailment

LTL Validity/Satisfiability

- Let ψ be an LTL formula

$$
\begin{aligned}
& \models \psi \quad \text{(LTL)} \\
\Longleftrightarrow\ & \neg\psi \text{ unsat} \\
\Longleftrightarrow\ & \mathcal{L}(A_{\neg\psi}) = \emptyset
\end{aligned}
$$

- $A_{\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\neg\psi$ (do not satisfy ψ)

LTL Entailment

- Let φ, ψ be an LTL formula

(LTL)

(LTL)

$\Longleftrightarrow$

$\Longleftrightarrow$

- $A_{\varphi\wedge\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\varphi \wedge \neg\psi$ (satisfy φ and do not satisfy ψ)

Automata-Theoretic LTL Satisfiability and Entailment

LTL Validity/Satisfiability

- Let ψ be an LTL formula

$$
\begin{aligned}
& \models \psi \quad \text{(LTL)} \\
\iff & \neg\psi \text{ unsat} \\
\iff & \mathcal{L}(A_{\neg\psi}) = \emptyset
\end{aligned}
$$

- $A_{\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\neg\psi$ (do not satisfy ψ)

LTL Entailment

- Let φ, ψ be an LTL formula

$$
\begin{aligned}
& \varphi \models \psi \quad \text{(LTL)} \\
& \models \varphi \rightarrow \psi \quad \text{(LTL)} \\
\iff & \varphi \wedge \neg\psi \text{ unsat} \\
\iff & \mathcal{L}(A_{\varphi\wedge\neg\psi}) = \emptyset
\end{aligned}
$$

- $A_{\varphi\wedge\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\varphi \wedge \neg\psi$ (satisfy φ and do not satisfy ψ)

Automata-Theoretic LTL Satisfiability and Entailment

LTL Validity/Satisfiability

- Let ψ be an LTL formula

$$
\begin{array}{ll}
 & \models \psi \quad \text{(LTL)} \\
\iff & \neg\psi \text{ unsat} \\
\iff & \mathcal{L}(A_{\neg\psi}) = \emptyset
\end{array}
$$

- $A_{\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\neg\psi$ (do not satisfy ψ)

LTL Entailment

- Let φ, ψ be an LTL formula

$$
\begin{array}{ll}
 & \varphi \models \psi \quad \text{(LTL)} \\
 & \models \varphi \rightarrow \psi \quad \text{(LTL)} \\
\iff & \varphi \wedge \neg\psi \text{ unsat} \\
\iff & \mathcal{L}(A_{\varphi\wedge\neg\psi}) = \emptyset
\end{array}
$$

- $A_{\varphi\wedge\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\varphi \wedge \neg\psi$ (satisfy φ and do not satisfy ψ)

Automata-Theoretic LTL Satisfiability and Entailment

LTL Validity/Satisfiability

- Let ψ be an LTL formula

$$\begin{array}{ll} & \models \psi \quad \text{(LTL)} \\ \Longleftrightarrow & \neg\psi \text{ unsat} \\ \Longleftrightarrow & \mathcal{L}(A_{\neg\psi}) = \emptyset \end{array}$$

- $A_{\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\neg\psi$ (do not satisfy ψ)

LTL Entailment

- Let φ, ψ be an LTL formula

$$\begin{array}{ll} & \varphi \models \psi \quad \text{(LTL)} \\ & \models \varphi \rightarrow \psi \quad \text{(LTL)} \\ \Longleftrightarrow & \varphi \wedge \neg\psi \text{ unsat} \\ \Longleftrightarrow & \mathcal{L}(A_{\varphi\wedge\neg\psi}) = \emptyset \end{array}$$

- $A_{\varphi\wedge\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\varphi \wedge \neg\psi$ (satisfy φ and do not satisfy ψ)

Automata-Theoretic LTL Satisfiability and Entailment

Two steps for checking $\models \psi$ [resp. $\varphi \models \psi$]

(i) Compute $A_{\neg\psi}$ [resp. $A_{\varphi\wedge\neg\psi}$]

(ii) Check the emptiness of $\mathcal{L}(A_{\neg\psi})$ [resp. $\mathcal{L}(A_{\varphi\wedge\neg\psi})$]

Automata-Theoretic LTL Satisfiability and Entailment

Two steps for checking $\models \psi$ [resp. $\varphi \models \psi$]

(i) Compute $A_{\neg\psi}$ [resp. $A_{\varphi\wedge\neg\psi}$]

(ii) Check the emptiness of $\mathcal{L}(A_{\neg\psi})$ [resp. $\mathcal{L}(A_{\varphi\wedge\neg\psi})$]

Automata-Theoretic LTL Satisfiability and Entailment

Two steps for checking $\models \psi$ [resp. $\varphi \models \psi$]

(i) Compute $A_{\neg\psi}$ [resp. $A_{\varphi\wedge\neg\psi}$]

(ii) Check the emptiness of $\mathcal{L}(A_{\neg\psi})$ [resp. $\mathcal{L}(A_{\varphi\wedge\neg\psi})$]

Automata-Theoretic LTL Model Checking

LTL Model Checking

- Let M be a Kripke model and ψ be an LTL formula

$$
\begin{aligned}
& M \models \psi \quad \text{(LTL)} \\
\Longleftrightarrow\ & \mathcal{L}(M) \subseteq \mathcal{L}(\psi) \\
\Longleftrightarrow\ & \mathcal{L}(M) \cap \overline{\mathcal{L}(\psi)} = \emptyset \\
\Longleftrightarrow\ & \mathcal{L}(M) \cap \mathcal{L}(\neg\psi) = \emptyset \\
\Longleftrightarrow\ & \mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\psi}) = \emptyset \\
\Longleftrightarrow\ & \mathcal{L}(A_M \times A_{\neg\psi}) = \emptyset
\end{aligned}
$$

- A_M is a Büchi Automaton equivalent to M (which represents all and only the executions of M)
- $A_{\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\neg\psi$ (do not satisfy ψ)

$\Longrightarrow$ $A_M \times A_{\neg\psi}$ represents all and only the paths appearing in M and not in ψ.

Automata-Theoretic LTL Model Checking

LTL Model Checking

- Let M be a Kripke model and ψ be an LTL formula

$$
\begin{array}{rl}
 & M \models \psi \quad \text{(LTL)} \\
\Longleftrightarrow & \mathcal{L}(M) \subseteq \underline{\mathcal{L}(\psi)} \\
\Longleftrightarrow & \mathcal{L}(M) \cap \overline{\mathcal{L}(\psi)} = \emptyset \\
\Longleftrightarrow & \mathcal{L}(M) \cap \mathcal{L}(\neg\psi) = \emptyset \\
\Longleftrightarrow & \mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\psi}) = \emptyset \\
\Longleftrightarrow & \mathcal{L}(A_M \times A_{\neg\psi}) = \emptyset
\end{array}
$$

- A_M is a Büchi Automaton equivalent to M (which represents all and only the executions of M)
- $A_{\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\neg\psi$ (do not satisfy ψ)

$\Longrightarrow$ $A_M \times A_{\neg\psi}$ represents all and only the paths appearing in M and not in ψ.

Automata-Theoretic LTL Model Checking

LTL Model Checking

- Let M be a Kripke model and ψ be an LTL formula

$$\begin{aligned}
& M \models \psi \quad \text{(LTL)} \\
\iff & \mathcal{L}(M) \subseteq \mathcal{L}(\psi) \\
\iff & \mathcal{L}(M) \cap \overline{\mathcal{L}(\psi)} = \emptyset \\
\iff & \mathcal{L}(M) \cap \mathcal{L}(\neg\psi) = \emptyset \\
\iff & \mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\psi}) = \emptyset \\
\iff & \mathcal{L}(A_M \times A_{\neg\psi}) = \emptyset
\end{aligned}$$

- A_M is a Büchi Automaton equivalent to M (which represents all and only the executions of M)
- $A_{\neg\psi}$ is a Büchi Automaton which represents all and only the paths that satisfy $\neg\psi$ (do not satisfy ψ)

$\Longrightarrow$ $A_M \times A_{\neg\psi}$ represents all and only the paths appearing in M and not in ψ.

Automata-Theoretic LTL Model Checking

Four steps

Let $\varphi \overset{\text{def}}{=} \neg\psi$:

(i) Compute A_M

(ii) Compute A_φ

(iii) Compute the product $A_M \times A_\varphi$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$

Automata-Theoretic LTL Model Checking

Four steps

Let $\varphi \overset{\text{def}}{=} \neg\psi$:

(i) Compute A_M

(ii) Compute A_φ

(iii) Compute the product $A_M \times A_\varphi$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$

Automata-Theoretic LTL Model Checking

Four steps

Let $\varphi \overset{\text{def}}{=} \neg\psi$:

(i) Compute A_M

(ii) Compute A_φ

(iii) Compute the product $A_M \times A_\varphi$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$

Automata-Theoretic LTL Model Checking

Four steps

Let $\varphi \overset{\text{def}}{=} \neg\psi$:

(i) Compute A_M

(ii) Compute A_φ

(iii) Compute the product $A_M \times A_\varphi$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$

Automata-Theoretic LTL Model Checking

Four steps

Let $\varphi \overset{\text{def}}{=} \neg\psi$:

(i) Compute A_M
(ii) Compute A_φ
(iii) Compute the product $A_M \times A_\varphi$
(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$

Outline

1 Büchi Automata

2 The Automata-Theoretic Approach to LTL Reasoning
  - General Ideas
  - Language-Emptiness Checking of Büchi Automata
  - From Kripke Models to Büchi Automata
  - From LTL Formulas to Büchi Automata
  - Complexity

3 Exercises

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
  - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
  (i) a DFS finds the accepting states *f* reachable from an initial state;
  (ii) for each *f*, a second DFS finds if it can reach f
  (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
  (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
  (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
  (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
  - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
  (i) a DFS finds the accepting states *f* reachable from an initial state;
  (ii) for each *f*, a second DFS finds if it can reach f
  (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
  (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
  (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
  (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
  - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
  (i) a DFS finds the accepting states *f* reachable from an initial state;
  (ii) for each *f*, a second DFS finds if it can reach f
  (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
  (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
  (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
  (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
  - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
  (i) a DFS finds the accepting states *f* reachable from an initial state;
  (ii) for each *f*, a second DFS finds if it can reach f
  (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
  (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
  (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
  (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
  - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
  (i) a DFS finds the accepting states *f* reachable from an initial state;
  (ii) for each *f*, a second DFS finds if it can reach f (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
  (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
  (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
  (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
    - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
    (i) a DFS finds the accepting states *f* reachable from an initial state;
    (ii) for each *f*, a second DFS finds if it can reach f (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
    (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
    (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
    (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
    - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
    (i) a DFS finds the accepting states f reachable from an initial state;
    (ii) for each f, a second DFS finds if it can reach f (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
    (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
    (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
    (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
  - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
  (i) a DFS finds the accepting states *f* reachable from an initial state;
  (ii) for each *f*, a second DFS finds if it can reach f (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
  (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
  (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
  (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
    - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
    (i) a DFS finds the accepting states *f* reachable from an initial state;
    (ii) for each *f*, a second DFS finds if it can reach f (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
    (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
    (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
    (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
  - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
  - (i) a DFS finds the accepting states f reachable from an initial state;
  - (ii) for each f, a second DFS finds if it can reach f (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
  - (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
  - (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
  - (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

NBA emptiness checking

- Idea: find an accepting cycle reachable from an initial state
    - accepting cycle: a cycle containing some accepting state f
- A naive algorithm (Naive Double Nested DFS algorithm):
    (i) a DFS finds the accepting states *f* reachable from an initial state;
    (ii) for each *f*, a second DFS finds if it can reach f (i.e., if there exists a loop)

  Complexity: $O(n^2)$
- SCC-based algorithm:
    (i) Tarjan's algorithm uses a DFS to find the SCCs in linear time;
    (ii) drop all SCCs which do not have at least one arc, and which do not contain at least one accepting state f
    (iii) another DFS finds if the union of non-trivial SCCs is reachable from an initial state.

  Complexity: $O(n)$
- Drawbacks: it stores too much information and does not find directly a counterexample.

(Smart) Double Nested DFS algorithm

(Smart) Double Nested DFS

- Two nested DFSs
    - `DFS1` finds the accepting states f reachable from an initial state
    - for each f, `DFS2` finds if it can reach f (i.e., if there exists a loop)
- Two Hash tables:
    - `T1`: reachable states
    - `T2`: states reachable from a reachable accepting state
- Two stacks:
    - `S1`: current branch of states reachable
    - `S2`: current branch of states reachable from accepting state f
- It stops as soon as it finds a counterexample.
- The counterexample is given by
    - the stack of `DFS2` (an accepting, preceded by cycle)
    - the stack of `DFS1` (a path from an initial state to the cycle)

- DFS1 invokes DFS2 on each f_i only after popping it (postorder)
- T2 passed by reference (or static) $\Longrightarrow$ is not reset at each call of DFS2 !

(Smart) Double Nested DFS algorithm

(Smart) Double Nested DFS

- Two nested DFSs
  - `DFS1` finds the accepting states f reachable from an initial state
  - for each f, `DFS2` finds if it can reach f (i.e., if there exists a loop)
- Two Hash tables:
  - `T1`: reachable states
  - `T2`: states reachable from a reachable accepting state
- Two stacks:
  - `S1`: current branch of states reachable
  - `S2`: current branch of states reachable from accepting state f
- It stops as soon as it finds a counterexample.
- The counterexample is given by
  - the stack of `DFS2` (an accepting, preceded by cycle)
  - the stack of `DFS1` (a path from an initial state to the cycle)

- DFS1 invokes DFS2 on each f_i only after popping it (postorder)
- T2 passed by reference (or static) $\Longrightarrow$ is not reset at each call of DFS2 !

(Smart) Double Nested DFS algorithm

(Smart) Double Nested DFS

- Two nested DFSs
  - `DFS1` finds the accepting states f reachable from an initial state
  - for each f, `DFS2` finds if it can reach f (i.e., if there exists a loop)
- Two Hash tables:
  - `T1`: reachable states
  - `T2`: states reachable from a reachable accepting state
- Two stacks:
  - `S1`: current branch of states reachable
  - `S2`: current branch of states reachable from accepting state f
- It stops as soon as it finds a counterexample.
- The counterexample is given by
  - the stack of `DFS2` (an accepting, preceded by cycle)
  - the stack of `DFS1` (a path from an initial state to the cycle)

- DFS1 invokes DFS2 on each f_i only after popping it (postorder)
- T2 passed by reference (or static) $\Longrightarrow$ is not reset at each call of DFS2 !

(Smart) Double Nested DFS algorithm

(Smart) Double Nested DFS

- Two nested DFSs
  - `DFS1` finds the accepting states f reachable from an initial state
  - for each f, `DFS2` finds if it can reach f (i.e., if there exists a loop)
- Two Hash tables:
  - `T1`: reachable states
  - `T2`: states reachable from a reachable accepting state
- Two stacks:
  - `S1`: current branch of states reachable
  - `S2`: current branch of states reachable from accepting state f
- It stops as soon as it finds a counterexample.
- The counterexample is given by
  - the stack of `DFS2` (an accepting, preceded by cycle)
  - the stack of `DFS1` (a path from an initial state to the cycle)

- DFS1 invokes DFS2 on each f_i only after popping it (postorder)
- T2 passed by reference (or static) $\Longrightarrow$ is not reset at each call of DFS2 !

(Smart) Double Nested DFS algorithm

(Smart) Double Nested DFS

- Two nested DFSs
  - `DFS1` finds the accepting states f reachable from an initial state
  - for each f, `DFS2` finds if it can reach f (i.e., if there exists a loop)
- Two Hash tables:
  - `T1`: reachable states
  - `T2`: states reachable from a reachable accepting state
- Two stacks:
  - `S1`: current branch of states reachable
  - `S2`: current branch of states reachable from accepting state f
- It stops as soon as it finds a counterexample.
- The counterexample is given by
  - the stack of `DFS2` (an accepting, preceded by cycle)
  - the stack of `DFS1` (a path from an initial state to the cycle)

- DFS1 invokes DFS2 on each f_i only after popping it (postorder)
- T2 passed by reference (or static) $\Longrightarrow$ is not reset at each call of DFS2 !

(Smart) Double Nested DFS algorithm

(Smart) Double Nested DFS

- Two nested DFSs
  - `DFS1` finds the accepting states f reachable from an initial state
  - for each f, `DFS2` finds if it can reach f (i.e., if there exists a loop)
- Two Hash tables:
  - `T1`: reachable states
  - `T2`: states reachable from a reachable accepting state
- Two stacks:
  - `S1`: current branch of states reachable
  - `S2`: current branch of states reachable from accepting state f
- It stops as soon as it finds a counterexample.
- The counterexample is given by
  - the stack of `DFS2` (an accepting, preceded by cycle)
  - the stack of `DFS1` (a path from an initial state to the cycle)

- DFS1 invokes DFS2 on each f_i only after popping it (postorder)
- T2 passed by reference (or static) $\Longrightarrow$ is not reset at each call of DFS2 !

(Smart) Double Nested DFS - First DFS

```
// returns True if empty language, false otherwise
Bool DFS1(NBA A) {
   stack S1=I; stack S2=∅;
   Hashtable T1=I; Hashtable T2=∅;
   while S1!=∅ {
      v=top(S1);
      if ∃w s.t. w∈ δ(v) && T1(w)==0 {
         hash(w,T1);
         push(w,S1);
      } else {
         pop(S1);
         if (v∈F && !DFS2(v,S2,T2,A))     //test after popping!
            return False;
   }     }
   return True;
}
```

(Smart) Double Nested DFS - Second DFS

```
Bool DFS2(state f, stack & S, Hashtable & T, NBA A) {
    hash(f,T);
    S = {f}
    while S!=$\emptyset$ {
        v=top(S);
        if f$\in\delta$(v) return False;
        if $\exists$w s.t. w$\in\delta$(v) && T(w)==0 {
            hash(w);
            push(w);
        } else pop(S);
    }
    return True;
}
```

Remark: `T` passed by reference (or static) $\Longrightarrow$ is not reset at each call of `DFS2` !

Double nested DFS: Intuition

DFS1 invokes DFS2 on each $f_1, \ldots, f_n$ only after popping it (postorder):

- suppose $DFS2$ is invoked on f_j earlier than on f_i

$\Longrightarrow$ f_i not reachable from (any state s which is reachable from) f_j

- If during $DFS2(f_i, \ldots)$ it is encountered a state S which has already been explored by $DFS2(f_j, \ldots)$ for some f_j,
  - can we reach f_i from S?
  - No, because f_i is not reachable from f_j!

$\Longrightarrow$ It is safe to backtrack!

Double nested DFS: Intuition

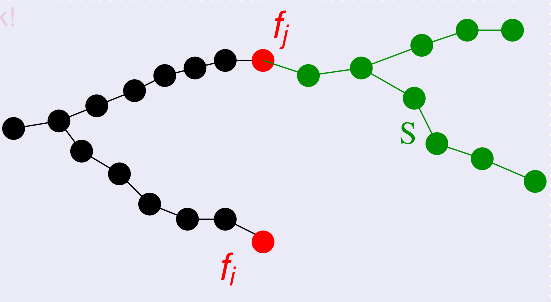
DFS1 invokes DFS2 on each $f_1, ..., f_n$ only after popping it (postorder):

- suppose $DFS2$ is invoked on f_j earlier than on f_i

$\Longrightarrow$ f_i not reachable from (any state s which is reachable from) f_j

- If during $DFS2(f_i, ...)$ it is encountered a state S which has already been explored by $DFS2(f_j, ...)$ for some f_j,
  - can we reach f_i from S?
  - No, because f_i is not reachable from f_j!

$\Longrightarrow$ It is safe to backtrack!



Double nested DFS: Intuition

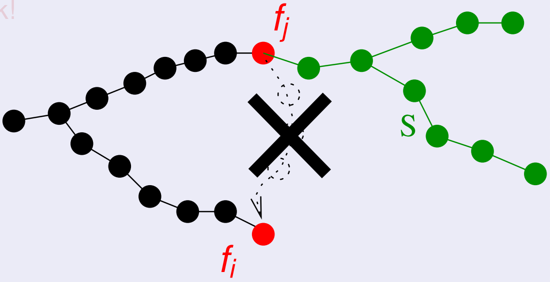
DFS1 invokes DFS2 on each $f_1, ..., f_n$ only after popping it (postorder):

- suppose *DFS*2 is invoked on f_j earlier than on f_i

$\Longrightarrow$ f_i not reachable from (any state s which is reachable from) f_j

- If during *DFS*2$(f_i, ...)$ it is encountered a state S which has already been explored by *DFS*2$(f_j, ...)$ for some f_j,
  - can we reach f_i from S?
  - No, because f_i is not reachable from f_j!

$\Longrightarrow$ It is safe to backtrack!



Double nested DFS: Intuition

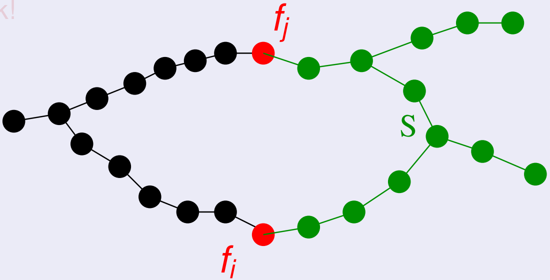
DFS1 invokes DFS2 on each $f_1, ..., f_n$ only after popping it (postorder):

- suppose *DFS*2 is invoked on f_j earlier than on f_i

$\Longrightarrow$ f_i not reachable from (any state s which is reachable from) f_j

- If during $DFS2(f_i, ...)$ it is encountered a state S which has already been explored by $DFS2(f_j, ...)$ for some f_j,
  - can we reach f_i from S?
  - No, because f_i is not reachable from f_j!

$\Longrightarrow$ It is safe to backtrack!



Double nested DFS: Intuition

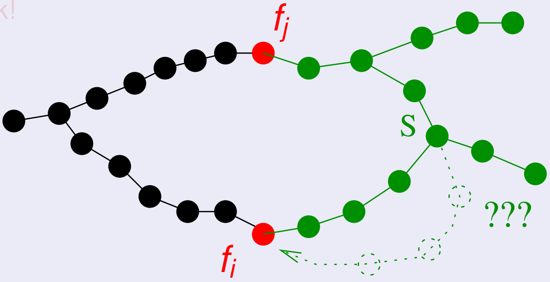
DFS1 invokes DFS2 on each $f_1, ..., f_n$ only after popping it (postorder):

- suppose *DFS*2 is invoked on f_j earlier than on f_i

$\Longrightarrow$ f_i not reachable from (any state s which is reachable from) f_j

- If during *DFS*2(f_i, ...) it is encountered a state S which has already been explored by *DFS*2(f_j, ...) for some f_j,
  - can we reach f_i from S?
  - No, because f_i is not reachable from f_j!

$\Longrightarrow$ It is safe to backtrack!



Double nested DFS: Intuition

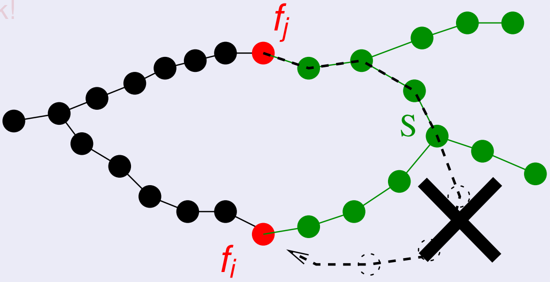
DFS1 invokes DFS2 on each $f_1, ..., f_n$ only after popping it (postorder):

- suppose *DFS*2 is invoked on f_j earlier than on f_i

⟹ f_i not reachable from (any state s which is reachable from) f_j

- If during *DFS*2(f_i, ...) it is encountered a state S which has already been explored by *DFS*2(f_j, ...) for some f_j,
  - can we reach f_i from S?
  - No, because f_i is not reachable from f_j!

⟹ It is safe to backtrack!



Double nested DFS: Intuition

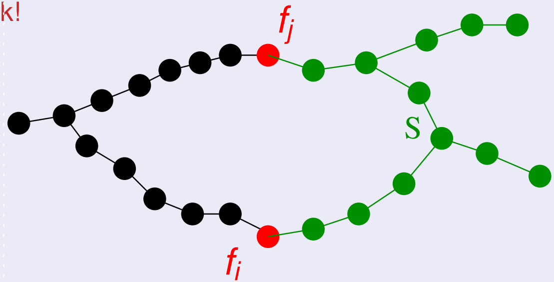
DFS1 invokes DFS2 on each $f_1, ..., f_n$ only after popping it (postorder):

- suppose *DFS*2 is invoked on f_j earlier than on f_i

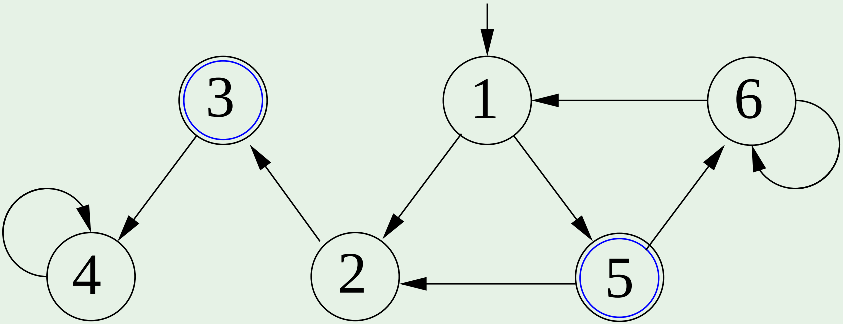
$\Longrightarrow$ f_i not reachable from (any state *s* which is reachable from) f_j

- If during $DFS2(f_i, ...)$ it is encountered a state *S* which has already been explored by $DFS2(f_j, ...)$ for some f_j,
  - can we reach f_i from *S*?
  - No, because f_i is not reachable from f_j!

$\Longrightarrow$ It is safe to backtrack!



(Smart) Double Nested DFS: example



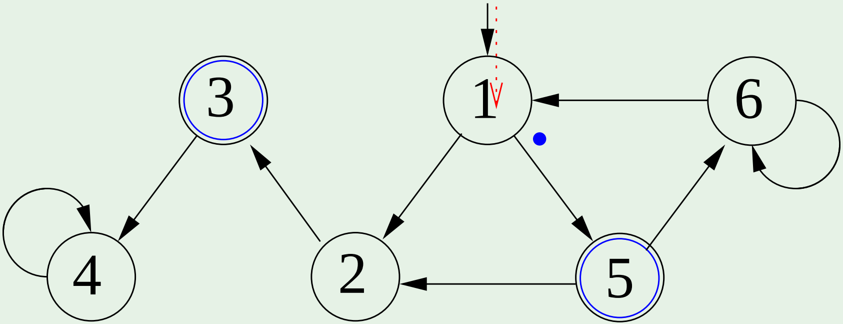
T1

S1

T2

S2

(Smart) Double Nested DFS: example



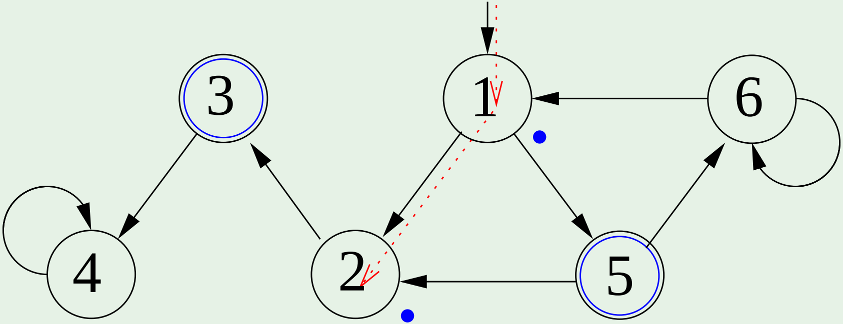
T1 1

S1 1

T2

S2

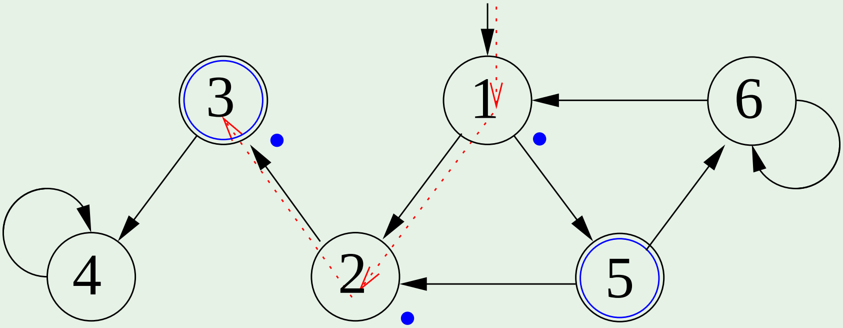
(Smart) Double Nested DFS: example



T1 1 2 T2

S1 1 2 S2

(Smart) Double Nested DFS: example



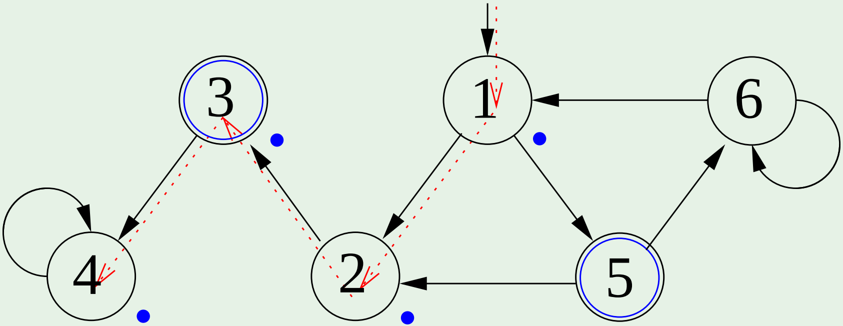
T1 1 2 3

S1 1 2 3

T2

S2

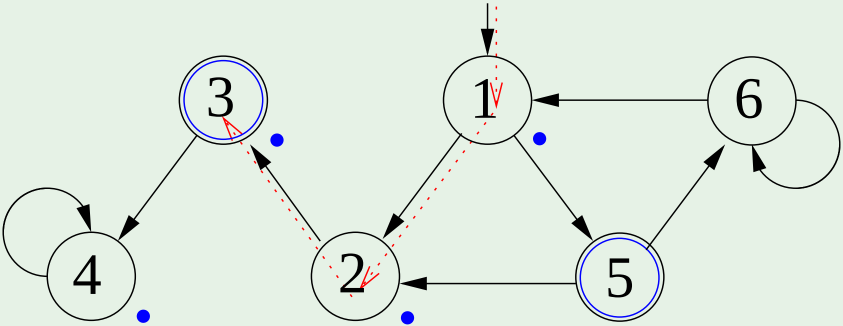
(Smart) Double Nested DFS: example



T1 1 2 3 4 T2

S1 1 2 3 4 S2

(Smart) Double Nested DFS: example



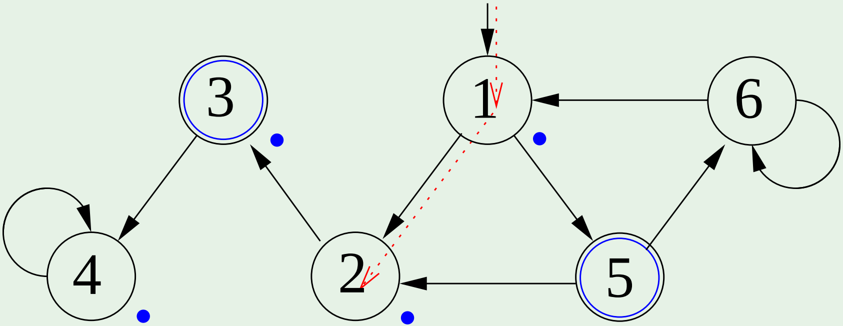
T1 1 2 3 4

S1 1 2 3

T2

S2

(Smart) Double Nested DFS: example



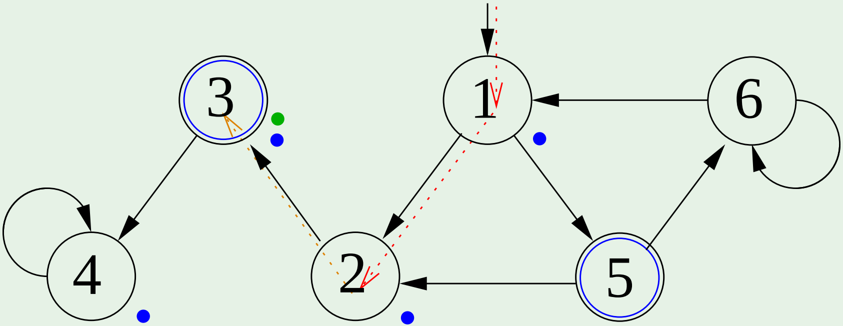
T1 1 2 3 4

S1 1 2

T2

S2

(Smart) Double Nested DFS: example



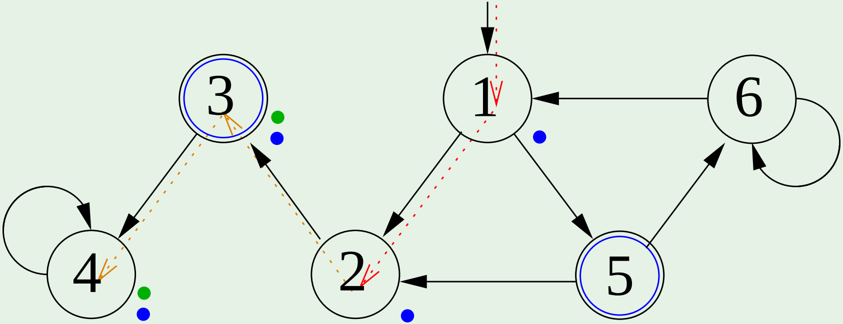
T1 1 2 3 4

T2 3

S1 1 2

S2 3

(Smart) Double Nested DFS: example



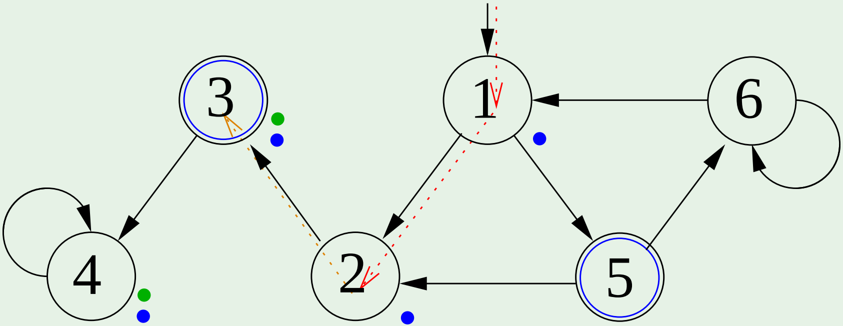
T1 1 2 3 4

T2 3 4

S1 1 2

S2 3 4

(Smart) Double Nested DFS: example



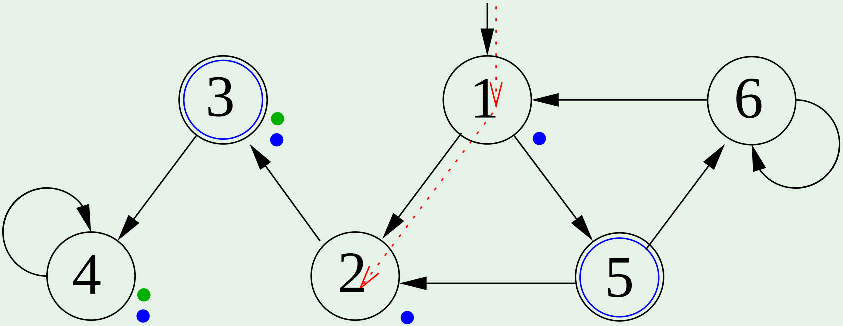
T1 1 2 3 4

S1 1 2

T2 3 4

S2 3

(Smart) Double Nested DFS: example



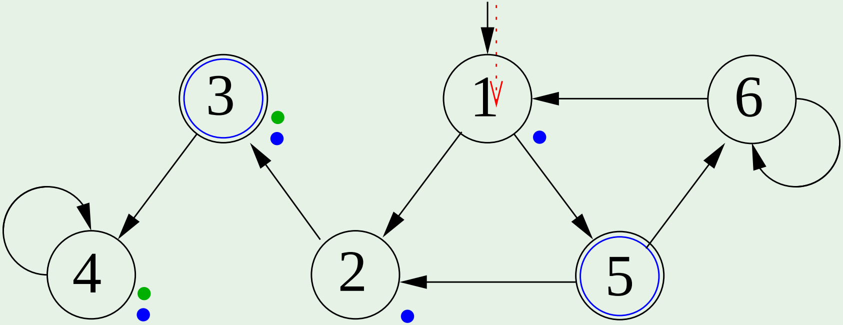
T1 1 2 3 4

S1 1 2

T2 3 4

S2

(Smart) Double Nested DFS: example



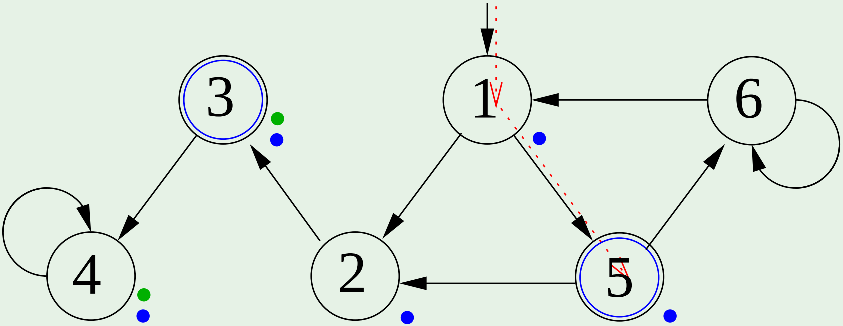
T1 1 2 3 4

T2 3 4

S1 1

S2

(Smart) Double Nested DFS: example



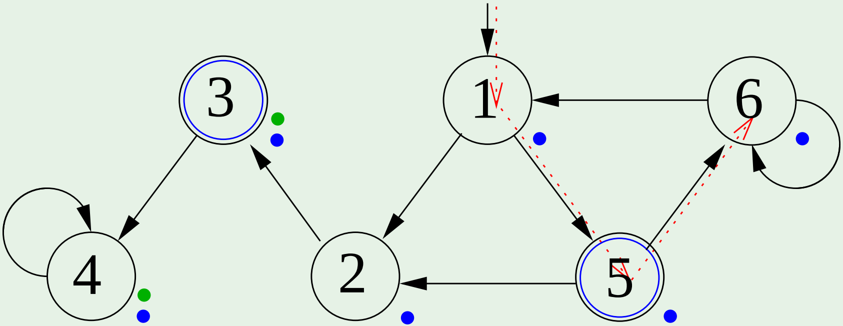
T1 1 2 3 4 5

T2 3 4

S1 1 5

S2

(Smart) Double Nested DFS: example



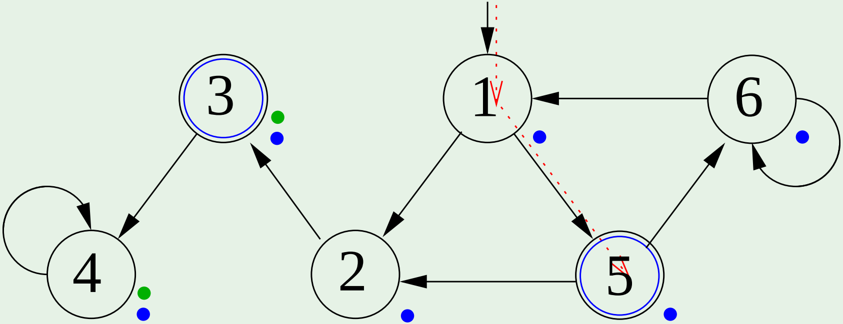
T1 1 2 3 4 5 6

T2 3 4

S1 1 5 6

S2

(Smart) Double Nested DFS: example



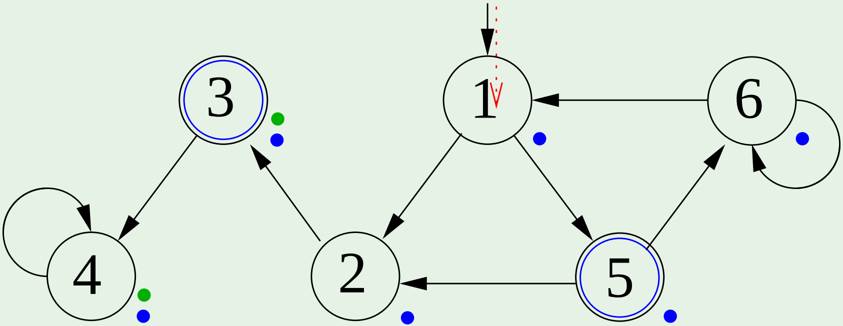
T1 1 2 3 4 5 6

T2 3 4

S1 1 5

S2

(Smart) Double Nested DFS: example



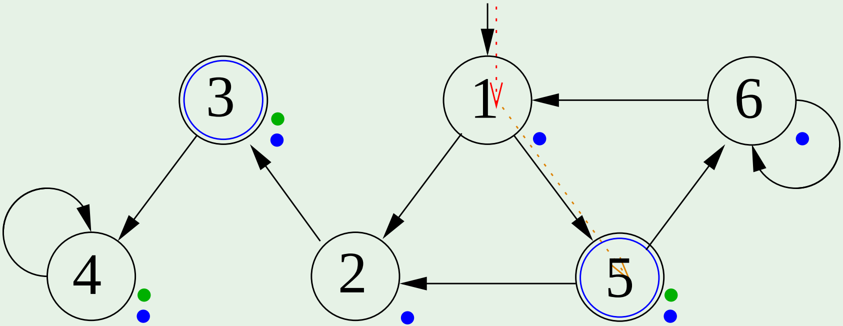
T1 1 2 3 4 5 6

T2 3 4

S1 1

S2

(Smart) Double Nested DFS: example



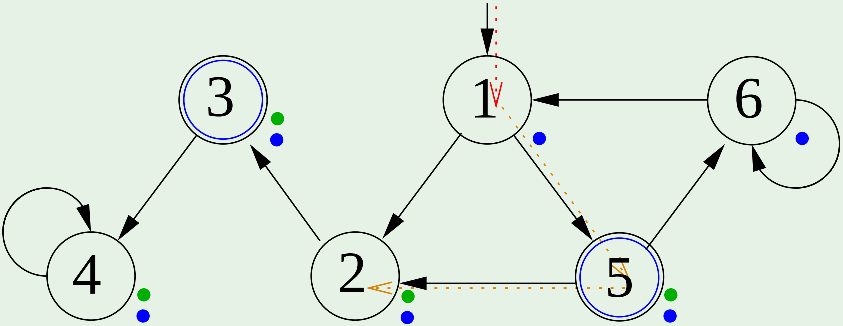
T1 1 2 3 4 5 6

S1 1

T2 3 4 5

S2 5

(Smart) Double Nested DFS: example



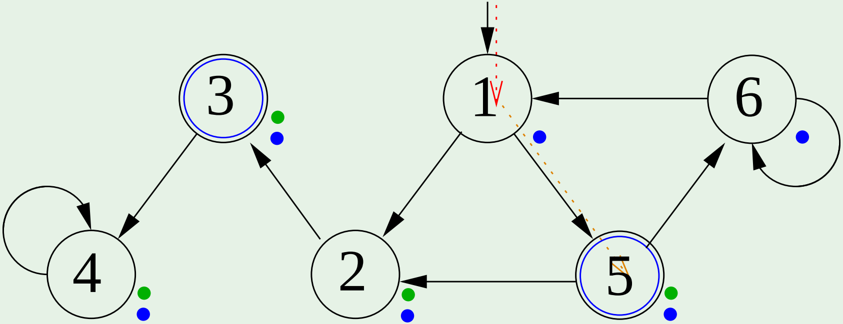
T1 1 2 3 4 5 6

T2 3 4 5 2

S1 1

S2 5 2

(Smart) Double Nested DFS: example



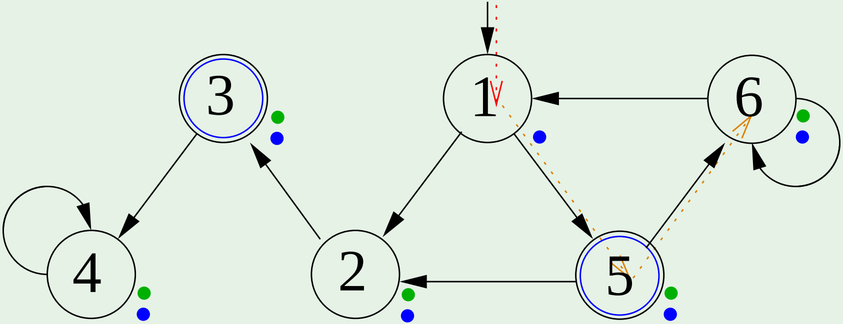
T1 1 2 3 4 5 6

S1 1

T2 3 4 5 2

S2 5

(Smart) Double Nested DFS: example



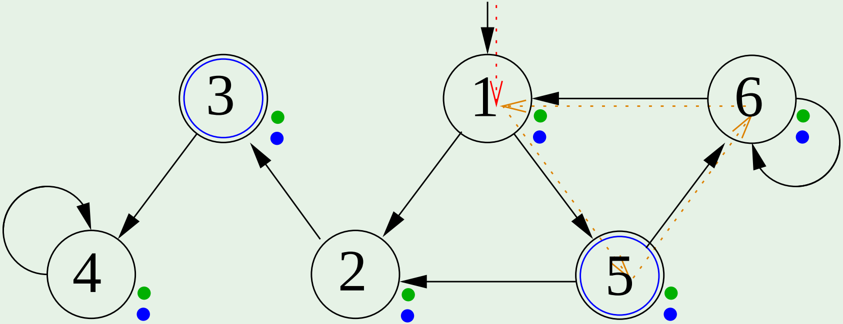
T1 1 2 3 4 5 6

T2 3 4 5 2 6

S1 1

S2 5 6

(Smart) Double Nested DFS: example



T1 1 2 3 4 5 6

T2 3 4 5 2 6 1

S1 1

S2 5 6 1

Outline

1. Büchi Automata

2. The Automata-Theoretic Approach to LTL Reasoning
   - General Ideas
   - Language-Emptiness Checking of Büchi Automata
   - From Kripke Models to Büchi Automata
   - From LTL Formulas to Büchi Automata
   - Complexity

3. Exercises

Computing an NBA A_M from a Kripke Structure M

- Transform a Kripke model $M = \langle S, S_0, R, L, AP \rangle$ into an NBA $A_M = \langle Q, \Sigma, \delta, I, F \rangle$ s.t.:
  - States: $Q := S \cup \{init\}$, *init* being a new initial state
  - Alphabet: $\Sigma := 2^{AP}$ (total truth-assignments as alphabet symbols!)
  - Initial State: $I := \{init\}$
  - Accepting States: $F := Q = S \cup \{init\}$
  - Transitions:

$$\begin{aligned} \delta: \quad & q \xrightarrow{a} q' \text{ iff } (q, q') \in R \; and \; L(q') = a \\ & init \xrightarrow{a} q \text{ iff } q \in S_0 \; and \; L(q) = a \end{aligned}$$

- $\mathcal{L}(A_M) = \mathcal{L}(M)$
- $|A_M| = |M| + 1$

Computing an NBA A_M from a Kripke Structure M

- Transform a Kripke model $M = \langle S, S_0, R, L, AP \rangle$ into an NBA $A_M = \langle Q, \Sigma, \delta, I, F \rangle$ s.t.:
  - States: $Q := S \cup \{init\}$, *init* being a new initial state
  - Alphabet: $\Sigma := 2^{AP}$ (total truth-assignments as alphabet symbols!)
  - Initial State: $I := \{init\}$
  - Accepting States: $F := Q = S \cup \{init\}$
  - Transitions:

$$\begin{aligned} \delta : \quad & q \xrightarrow{a} q' \text{ iff } (q, q') \in R \text{ and } L(q') = a \\ & init \xrightarrow{a} q \text{ iff } q \in S_0 \text{ and } L(q) = a \end{aligned}$$

- $\mathcal{L}(A_M) = \mathcal{L}(M)$
- $|A_M| = |M| + 1$

Computing an NBA A_M from a Kripke Structure M

- Transform a Kripke model $M = \langle S, S_0, R, L, AP \rangle$ into an NBA $A_M = \langle Q, \Sigma, \delta, I, F \rangle$ s.t.:
  - States: $Q := S \cup \{init\}$, *init* being a new initial state
  - Alphabet: $\Sigma := 2^{AP}$ (total truth-assignments as alphabet symbols!)
  - Initial State: $I := \{init\}$
  - Accepting States: $F := Q = S \cup \{init\}$
  - Transitions:

$$\begin{aligned} \delta : \quad & q \xrightarrow{a} q' \text{ iff } (q, q') \in R \text{ and } L(q') = a \\ & init \xrightarrow{a} q \text{ iff } q \in S_0 \text{ and } L(q) = a \end{aligned}$$

- $\mathcal{L}(A_M) = \mathcal{L}(M)$
- $|A_M| = |M| + 1$

Computing an NBA A_M from a Kripke Structure M

- Transform a Kripke model $M = \langle S, S_0, R, L, AP \rangle$ into an NBA $A_M = \langle Q, \Sigma, \delta, I, F \rangle$ s.t.:
  - States: $Q := S \cup \{init\}$, *init* being a new initial state
  - Alphabet: $\Sigma := 2^{AP}$ (total truth-assignments as alphabet symbols!)
  - Initial State: $I := \{init\}$
  - Accepting States: $F := Q = S \cup \{init\}$
  - Transitions:

$$\delta : \quad \begin{array}{l} q \xrightarrow{a} q' \text{ iff } (q, q') \in R \textit{ and } L(q') = a \\ init \xrightarrow{a} q \text{ iff } q \in S_0 \textit{ and } L(q) = a \end{array}$$

- $\mathcal{L}(A_M) = \mathcal{L}(M)$
- $|A_M| = |M| + 1$

Computing an NBA A_M from a Kripke Structure M

- Transform a Kripke model $M = \langle S, S_0, R, L, AP \rangle$ into an NBA $A_M = \langle Q, \Sigma, \delta, I, F \rangle$ s.t.:
  - States: $Q := S \cup \{init\}$, *init* being a new initial state
  - Alphabet: $\Sigma := 2^{AP}$ (total truth-assignments as alphabet symbols!)
  - Initial State: $I := \{init\}$
  - Accepting States: $F := Q = S \cup \{init\}$
  - Transitions:

$$\begin{aligned} \delta : \quad & q \xrightarrow{a} q' \text{ iff } (q, q') \in R \; and \; L(q') = a \\ & init \xrightarrow{a} q \text{ iff } q \in S_0 \; and \; L(q) = a \end{aligned}$$

- $\mathcal{L}(A_M) = \mathcal{L}(M)$
- $|A_M| = |M| + 1$

Computing an NBA A_M from a Kripke Structure M

- Transform a Kripke model $M = \langle S, S_0, R, L, AP \rangle$ into an NBA $A_M = \langle Q, \Sigma, \delta, I, F \rangle$ s.t.:
  - States: $Q := S \cup \{init\}$, *init* being a new initial state
  - Alphabet: $\Sigma := 2^{AP}$ (total truth-assignments as alphabet symbols!)
  - Initial State: $I := \{init\}$
  - Accepting States: $F := Q = S \cup \{init\}$
  - Transitions:

$$\begin{array}{ll}\delta: & q \xrightarrow{a} q' \text{ iff } (q, q') \in R \textit{ and } L(q') = a \\ & init \xrightarrow{a} q \text{ iff } q \in S_0 \textit{ and } L(q) = a\end{array}$$

- $\mathcal{L}(A_M) = \mathcal{L}(M)$
- $|A_M| = |M| + 1$

Computing an NBA A_M from a Kripke Structure M

- Transform a Kripke model $M = \langle S, S_0, R, L, AP \rangle$ into an NBA $A_M = \langle Q, \Sigma, \delta, I, F \rangle$ s.t.:
  - States: $Q := S \cup \{init\}$, *init* being a new initial state
  - Alphabet: $\Sigma := 2^{AP}$ (total truth-assignments as alphabet symbols!)
  - Initial State: $I := \{init\}$
  - Accepting States: $F := Q = S \cup \{init\}$
  - Transitions:

$$\begin{array}{ll} \delta : & q \xrightarrow{a} q' \text{ iff } (q, q') \in R \textit{ and } L(q') = a \\ & init \xrightarrow{a} q \text{ iff } q \in S_0 \textit{ and } L(q) = a \end{array}$$

- $\mathcal{L}(A_M) = \mathcal{L}(M)$
- $|A_M| = |M| + 1$

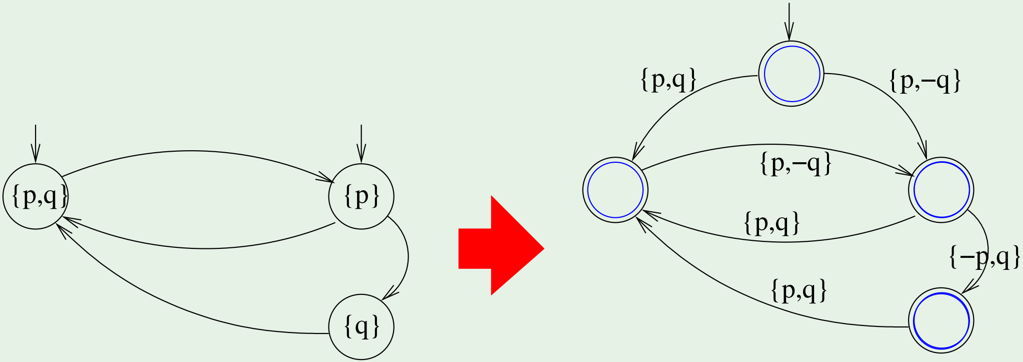
Computing an NBA A_M from a Kripke Structure M

- Transform a Kripke model $M = \langle S, S_0, R, L, AP \rangle$ into an NBA $A_M = \langle Q, \Sigma, \delta, I, F \rangle$ s.t.:
  - States: $Q := S \cup \{init\}$, *init* being a new initial state
  - Alphabet: $\Sigma := 2^{AP}$ (total truth-assignments as alphabet symbols!)
  - Initial State: $I := \{init\}$
  - Accepting States: $F := Q = S \cup \{init\}$
  - Transitions:

$$\begin{array}{ll} \delta : & q \xrightarrow{a} q' \text{ iff } (q, q') \in R \textit{ and } L(q') = a \\ & init \xrightarrow{a} q \text{ iff } q \in S_0 \textit{ and } L(q) = a \end{array}$$

- $\mathcal{L}(A_M) = \mathcal{L}(M)$
- $|A_M| = |M| + 1$

Computing a NBA A_M from a Kripke Structure M: Example



Kripke Structure

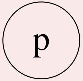
Buechi Automaton

$\Longrightarrow$ Substantially:

1. add one initial state,
2. move labels from states to incoming edges,
3. set all states as accepting states

Labels on Kripke Structures and BA's - Remark

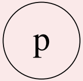
Note that the labels of a Büchi Automaton are different from the labels of a Kripke Structure. Also graphically, they are interpreted differently:



- in a Kripke Structure, it means that *p* is true and all other propositions are false;
- in a Büchi Automaton, it means that *p* is true and all other propositions are irrelevant ("don't care"), i.e. they can be either true or false.

Labels on Kripke Structures and BA's - Remark

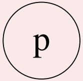
Note that the labels of a Büchi Automaton are different from the labels of a Kripke Structure. Also graphically, they are interpreted differently:



- in a Kripke Structure, it means that *p* is true and all other propositions are false;
- in a Büchi Automaton, it means that *p* is true and all other propositions are irrelevant ("don't care"), i.e. they can be either true or false.

Labels on Kripke Structures and BA's - Remark

Note that the labels of a Büchi Automaton are different from the labels of a Kripke Structure. Also graphically, they are interpreted differently:



- in a Kripke Structure, it means that *p* is true and all other propositions are false;
- in a Büchi Automaton, it means that *p* is true and all other propositions are irrelevant ("don't care"), i.e. they can be either true or false.

Outline

1. Büchi Automata

2. The Automata-Theoretic Approach to LTL Reasoning
   - General Ideas
   - Language-Emptiness Checking of Büchi Automata
   - From Kripke Models to Büchi Automata
   - From LTL Formulas to Büchi Automata
   - Complexity

3. Exercises

Translation problem

Problem

Given an LTL formula ϕ, find a Büchi Automaton that accepts the same language of ϕ.

- It is a fundamental problem in LTL validity/satisfiability/entailment and model checking
- We translate an LTL formula into a Generalized Büchi Automata (GBA), then into an NBA

Translation problem

Problem

Given an LTL formula ϕ, find a Büchi Automaton that accepts the same language of ϕ.

- It is a fundamental problem in LTL validity/satisfiability/entailment and model checking
- We translate an LTL formula into a Generalized Büchi Automata (GBA), then into an NBA

Translation problem

Problem

Given an LTL formula ϕ, find a Büchi Automaton that accepts the same language of ϕ.

- It is a fundamental problem in LTL validity/satisfiability/entailment and model checking
- We translate an LTL formula into a Generalized Büchi Automata (GBA), then into an NBA

LTL Negative Normal Form (NNF)

- Every LTL formula φ can be written into an equivalent formula φ' using only the operators $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$, $\mathbf{R}$ on propositional literals.
- Done by pushing negations down to literal level:

$$\begin{array}{lcl}
\neg\neg\varphi_1 & \Longrightarrow & \varphi_1 \\
\neg(\varphi_1 \vee \varphi_2) & \Longrightarrow & (\neg\varphi_1 \wedge \neg\varphi_2) \\
\neg(\varphi_1 \wedge \varphi_2) & \Longrightarrow & (\neg\varphi_1 \vee \neg\varphi_2) \\
\neg\mathbf{X}\varphi_1 & \Longrightarrow & \mathbf{X}\neg\varphi_1 \\
\neg(\varphi_1\mathbf{U}\varphi_2) & \Longrightarrow & (\neg\varphi_1\mathbf{R}\neg\varphi_2) \\
\neg(\varphi_1\mathbf{R}\varphi_2) & \Longrightarrow & (\neg\varphi_1\mathbf{U}\neg\varphi_2)
\end{array}$$

$\Longrightarrow$ The resulting formula is expressed in terms of $\vee$, $\wedge$, X, **U**, **R** and literals (Negative Normal Form, NNF).

  - the encoding is linear if a DAG representation is used

- In the construction of A_φ we now assume that φ is in NNF.
  $\Longrightarrow$ every non-atomic subformula occurs positively in φ
- For convenience, we still use **F**'s and **G**'s as shortcuts: $\mathbf{F}\varphi$ for $\top\mathbf{U}\varphi$ and $\mathbf{G}\varphi$ for $\bot\mathbf{R}\varphi$

LTL Negative Normal Form (NNF)

- Every LTL formula φ can be written into an equivalent formula φ' using only the operators $\wedge$, $\vee$, **X**, **U**, **R** on propositional literals.
- Done by pushing negations down to literal level:

$$\begin{array}{lcl}
\neg\neg\varphi_1 & \Longrightarrow & \varphi_1 \\
\neg(\varphi_1 \vee \varphi_2) & \Longrightarrow & (\neg\varphi_1 \wedge \neg\varphi_2) \\
\neg(\varphi_1 \wedge \varphi_2) & \Longrightarrow & (\neg\varphi_1 \vee \neg\varphi_2) \\
\neg\mathbf{X}\varphi_1 & \Longrightarrow & \mathbf{X}\neg\varphi_1 \\
\neg(\varphi_1\mathbf{U}\varphi_2) & \Longrightarrow & (\neg\varphi_1\mathbf{R}\neg\varphi_2) \\
\neg(\varphi_1\mathbf{R}\varphi_2) & \Longrightarrow & (\neg\varphi_1\mathbf{U}\neg\varphi_2)
\end{array}$$

$\Longrightarrow$ The resulting formula is expressed in terms of $\vee$, $\wedge$, X, **U**, **R** and literals (Negative Normal Form, NNF).

  - the encoding is linear if a DAG representation is used

- In the construction of A_φ we now assume that φ is in NNF.
  $\Longrightarrow$ every non-atomic subformula occurs positively in φ
- For convenience, we still use **F**'s and **G**'s as shortcuts: $\mathbf{F}\varphi$ for $\top\mathbf{U}\varphi$ and $\mathbf{G}\varphi$ for $\bot\mathbf{R}\varphi$

LTL Negative Normal Form (NNF)

- Every LTL formula φ can be written into an equivalent formula φ' using only the operators $\wedge$, $\vee$, **X**, **U**, **R** on propositional literals.
- Done by pushing negations down to literal level:

$$\begin{array}{lcl}
\neg\neg\varphi_1 & \Longrightarrow & \varphi_1 \\
\neg(\varphi_1 \vee \varphi_2) & \Longrightarrow & (\neg\varphi_1 \wedge \neg\varphi_2) \\
\neg(\varphi_1 \wedge \varphi_2) & \Longrightarrow & (\neg\varphi_1 \vee \neg\varphi_2) \\
\neg\mathbf{X}\varphi_1 & \Longrightarrow & \mathbf{X}\neg\varphi_1 \\
\neg(\varphi_1\mathbf{U}\varphi_2) & \Longrightarrow & (\neg\varphi_1\mathbf{R}\neg\varphi_2) \\
\neg(\varphi_1\mathbf{R}\varphi_2) & \Longrightarrow & (\neg\varphi_1\mathbf{U}\neg\varphi_2)
\end{array}$$

$\Longrightarrow$ The resulting formula is expressed in terms of $\vee$, $\wedge$, X, **U**, **R** and literals (Negative Normal Form, NNF).

  - the encoding is linear if a DAG representation is used

- In the construction of A_φ we now assume that φ is in NNF.
  $\Longrightarrow$ every non-atomic subformula occurs positively in φ
- For convenience, we still use **F**'s and **G**'s as shortcuts: $\mathbf{F}\varphi$ for $\top\mathbf{U}\varphi$ and $\mathbf{G}\varphi$ for $\bot\mathbf{R}\varphi$

LTL Negative Normal Form (NNF)

- Every LTL formula φ can be written into an equivalent formula φ' using only the operators $\wedge$, $\vee$, **X**, **U**, **R** on propositional literals.
- Done by pushing negations down to literal level:

$$\begin{array}{lcl}
\neg\neg\varphi_1 & \Longrightarrow & \varphi_1 \\
\neg(\varphi_1 \vee \varphi_2) & \Longrightarrow & (\neg\varphi_1 \wedge \neg\varphi_2) \\
\neg(\varphi_1 \wedge \varphi_2) & \Longrightarrow & (\neg\varphi_1 \vee \neg\varphi_2) \\
\neg\mathbf{X}\varphi_1 & \Longrightarrow & \mathbf{X}\neg\varphi_1 \\
\neg(\varphi_1\mathbf{U}\varphi_2) & \Longrightarrow & (\neg\varphi_1\mathbf{R}\neg\varphi_2) \\
\neg(\varphi_1\mathbf{R}\varphi_2) & \Longrightarrow & (\neg\varphi_1\mathbf{U}\neg\varphi_2)
\end{array}$$

$\Longrightarrow$ The resulting formula is expressed in terms of $\vee$, $\wedge$, X, **U**, **R** and literals (Negative Normal Form, NNF).

  - the encoding is linear if a DAG representation is used

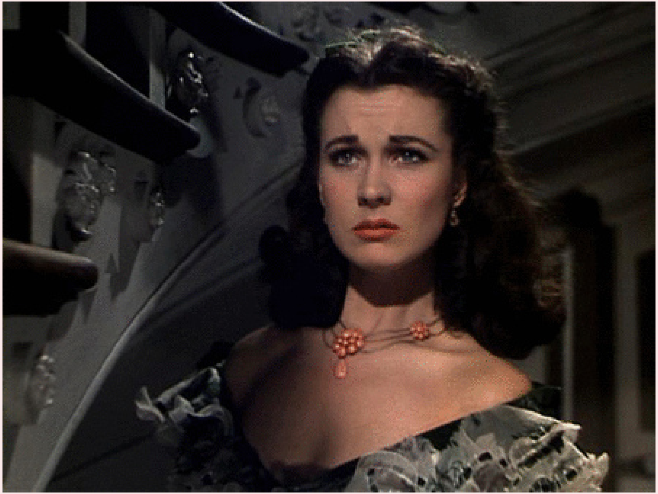
- In the construction of A_φ we now assume that φ is in NNF.
  $\Longrightarrow$ every non-atomic subformula occurs positively in φ
- For convenience, we still use **F**'s and **G**'s as shortcuts: $\mathbf{F}\varphi$ for $\top\mathbf{U}\varphi$ and $\mathbf{G}\varphi$ for $\bot\mathbf{R}\varphi$

On-the-fly Construction of A_φ (Intuition)

(Implicitly) Apply recursively the following steps:

Step 1: Apply the tableau expansion rules to φ:
$\psi_1 \mathbf{U} \psi_2 \Longrightarrow \psi_2 \vee (\psi_1 \wedge \mathbf{X}(\psi_1 \mathbf{U} \psi_2))$ [and $\mathbf{F}\psi \Longrightarrow \psi \vee \mathbf{XF}\psi$]
$\psi_1 \mathbf{R} \psi_2 \Longrightarrow \psi_2 \wedge (\psi_1 \vee \mathbf{X}(\psi_1 \mathbf{R} \psi_2))$ [and $\mathbf{G}\psi \Longrightarrow \psi \wedge \mathbf{XG}\psi$]
until we get a Boolean combination of elementary subformulas of φ
(An elementary formula is a proposition or a **X**-formula.)

Tableaux Rules: a Quote



“After all... tomorrow is another day."
[Scarlett O’Hara, “Gone with the Wind”]

On-the-fly Construction of A_φ (Intuition) [cont.]

Step 2: Convert all formulas into Disjunctive Normal Form, by:
(i) applying recursively the DeMorgan rule: $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \Longrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$, and then
(ii) pushing the conjunctions inside the next operator:

$$\varphi \xRightarrow{(i)} \bigvee_i (\bigwedge_j l_{ij} \wedge \bigwedge_k \mathbf{X} \psi_{ik}) \xRightarrow{(ii)} \bigvee_i (\bigwedge_j l_{ij} \wedge \mathbf{X} \bigwedge_k \psi_{ik}).$$

- Each disjunct $(\overbrace{\bigwedge_j l_{ij}}^{labels} \wedge \overbrace{\mathbf{X} \bigwedge_k \psi_{ik}}^{next\ part})$ represents a state:
  - the conjunction of literals $\bigwedge_j l_{ij}$ represents a set of labels in Σ (e.g., if $Vars(\varphi) = \{p, q, r\}$, $p \wedge \neg q$ represents the two labels $\{p, \neg q, r\}$ and $\{p, \neg q, \neg r\}$)
  - $\mathbf{X} \bigwedge_k \psi_{ik}$ represents the next part of the state (obbligations for the successors)
- N.B., if no next part occurs, $\mathbf{X}\top$ is implicitly assumed

On-the-fly Construction of A_φ (Intuition) [cont.]

Step 2: Convert all formulas into Disjunctive Normal Form, by:
(i) applying recursively the DeMorgan rule: $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \Longrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$, and then
(ii) pushing the conjunctions inside the next operator:

$$\varphi \overset{(i)}{\Longrightarrow} \bigvee_i \left(\bigwedge_j l_{ij} \wedge \bigwedge_k \mathbf{X}\psi_{ik}\right) \overset{(ii)}{\Longrightarrow} \bigvee_i \left(\bigwedge_j l_{ij} \wedge \mathbf{X} \bigwedge_k \psi_{ik}\right).$$

- Each disjunct $(\overbrace{\bigwedge_j l_{ij}}^{labels} \wedge \overbrace{\mathbf{X} \bigwedge_k \psi_{ik}}^{next\ part})$ represents a state:
  - the conjunction of literals $\bigwedge_j l_{ij}$ represents a set of labels in Σ (e.g., if $Vars(\varphi) = \{p, q, r\}$, $p \wedge \neg q$ represents the two labels $\{p, \neg q, r\}$ and $\{p, \neg q, \neg r\}$)
  - $\mathbf{X} \bigwedge_k \psi_{ik}$ represents the next part of the state (obbligations for the successors)
- N.B., if no next part occurs, $\mathbf{X}\top$ is implicitly assumed

On-the-fly Construction of A_φ (Intuition) [cont.]

Step 2: Convert all formulas into Disjunctive Normal Form, by:
(i) applying recursively the DeMorgan rule: $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \Longrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$, and then
(ii) pushing the conjunctions inside the next operator:

$$\varphi \overset{(i)}{\Longrightarrow} \bigvee_i (\bigwedge_j l_{ij} \wedge \bigwedge_k \mathbf{X}\psi_{ik}) \overset{(ii)}{\Longrightarrow} \bigvee_i (\bigwedge_j l_{ij} \wedge \mathbf{X} \bigwedge_k \psi_{ik}).$$

- Each disjunct $(\overbrace{\bigwedge_j l_{ij}}^{labels} \wedge \overbrace{\mathbf{X} \bigwedge_k \psi_{ik}}^{next\ part})$ represents a state:
  - the conjunction of literals $\bigwedge_j l_{ij}$ represents a set of labels in Σ
    (e.g., if $Vars(\varphi) = \{p, q, r\}$, $p \wedge \neg q$ represents the two labels $\{p, \neg q, r\}$ and $\{p, \neg q, \neg r\}$)
  - $\mathbf{X} \bigwedge_k \psi_{ik}$ represents the next part of the state
    (obbligations for the successors)
- N.B., if no next part occurs, $\mathbf{X}\top$ is implicitly assumed

On-the-fly Construction of A_φ (Intuition) [cont.]

Step 2: Convert all formulas into Disjunctive Normal Form, by:
(i) applying recursively the DeMorgan rule: $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \implies (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$, and then
(ii) pushing the conjunctions inside the next operator:

$$\varphi \overset{(i)}{\Longrightarrow} \bigvee_i (\bigwedge_j l_{ij} \wedge \bigwedge_k \mathbf{X} \psi_{ik}) \overset{(ii)}{\Longrightarrow} \bigvee_i (\bigwedge_j l_{ij} \wedge \mathbf{X} \bigwedge_k \psi_{ik}).$$

- Each disjunct $(\overbrace{\bigwedge_j l_{ij}}^{labels} \wedge \overbrace{\mathbf{X} \bigwedge_k \psi_{ik}}^{next\ part})$ represents a state:
  - the conjunction of literals $\bigwedge_j l_{ij}$ represents a set of labels in Σ
    (e.g., if $Vars(\varphi) = \{p, q, r\}$, $p \wedge \neg q$ represents the two labels $\{p, \neg q, r\}$ and $\{p, \neg q, \neg r\}$)
  - $\mathbf{X} \bigwedge_k \psi_{ik}$ represents the next part of the state
    (obbligations for the successors)
- N.B., if no next part occurs, $\mathbf{X}\top$ is implicitly assumed

On-the-fly Construction of A_φ (Intuition) [cont.]

Step 2: Convert all formulas into Disjunctive Normal Form, by:
(i) applying recursively the DeMorgan rule: $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \Longrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$, and then
(ii) pushing the conjunctions inside the next operator:

$$\varphi \overset{(i)}{\Longrightarrow} \bigvee_i (\bigwedge_j l_{ij} \wedge \bigwedge_k \mathbf{X}\psi_{ik}) \overset{(ii)}{\Longrightarrow} \bigvee_i (\bigwedge_j l_{ij} \wedge \mathbf{X} \bigwedge_k \psi_{ik}).$$

- Each disjunct $(\overbrace{\bigwedge_j l_{ij}}^{labels} \wedge \overbrace{\mathbf{X} \bigwedge_k \psi_{ik}}^{next\ part})$ represents a state:
  - the conjunction of literals $\bigwedge_j l_{ij}$ represents a set of labels in Σ
    (e.g., if $Vars(\varphi) = \{p, q, r\}$, $p \wedge \neg q$ represents the two labels $\{p, \neg q, r\}$ and $\{p, \neg q, \neg r\}$)
  - $\mathbf{X} \bigwedge_k \psi_{ik}$ represents the next part of the state
    (obbligations for the successors)
- N.B., if no next part occurs, $\mathbf{X}\top$ is implicitly assumed

On-the-fly Construction of A_φ (Intuition) [cont.]

Step 3: For every state S_i represented by $(\bigwedge_j l_{ij} \wedge \mathbf{X} \overbrace{\bigwedge_k \psi_{ik}}^{\varphi_i})$

- label the incoming edges of S_i with $\bigwedge_j l_{ij}$
- mark that the state S_i satisfies φ
- apply recursively steps 1-2-3 to $\varphi_i \stackrel{\text{def}}{=} \bigwedge_k \psi_{ik}$,
  - rewrite φ_i into $\bigvee_{i'}(\bigwedge_j l'_{i'j} \wedge \mathbf{X} \bigwedge_k \psi'_{i'k})$
  - from each disjunct $(\bigwedge_j l'_{i'j} \wedge \mathbf{X} \bigwedge_k \psi'_{i'k})$ generate a new state $S_{ii'}$ (if not already present) and label it as satisfying $\varphi_i \stackrel{\text{def}}{=} \bigwedge_k \psi_{ik}$
- draw an edge from S_i to all states $S_{ii'}$ which satisfy $\bigwedge_k \psi_{ik}$
- (if no next part occurs, $\mathbf{X}\top$ is implicitly assumed, so that an edge to a "true" node is drawn)

On-the-fly Construction of A_φ (Intuition) [cont.]

Step 3: For every state S_i represented by $(\bigwedge_j l_{ij} \land \mathbf{X} \overbrace{\bigwedge_k \psi_{ik}}^{\varphi_i})$

- label the incoming edges of S_i with $\bigwedge_j l_{ij}$
- mark that the state S_i satisfies φ
- apply recursively steps 1-2-3 to $\varphi_i \stackrel{\text{def}}{=} \bigwedge_k \psi_{ik}$,
  - rewrite φ_i into $\bigvee_{i'}(\bigwedge_j l'_{i'j} \land \mathbf{X} \bigwedge_k \psi'_{i'k})$
  - from each disjunct $(\bigwedge_j l'_{i'j} \land \mathbf{X} \bigwedge_k \psi'_{i'k})$ generate a new state $S_{ii'}$ (if not already present) and label it as satisfying $\varphi_i \stackrel{\text{def}}{=} \bigwedge_k \psi_{ik}$
- draw an edge from S_i to all states $S_{ii'}$ which satisfy $\bigwedge_k \psi_{ik}$
- (if no next part occurs, $\mathbf{X}\top$ is implicitly assumed, so that an edge to a “true” node is drawn)

On-the-fly Construction of A_φ (Intuition) [cont.]

Step 3: For every state S_i represented by $(\bigwedge_j l_{ij} \wedge \mathbf{X} \overbrace{\bigwedge_k \psi_{ik}}^{\varphi_i})$

- label the incoming edges of S_i with $\bigwedge_j l_{ij}$
- mark that the state S_i satisfies φ
- apply recursively steps 1-2-3 to $\varphi_i \stackrel{\text{def}}{=} \bigwedge_k \psi_{ik}$,
  - rewrite φ_i into $\bigvee_{i'} (\bigwedge_j l'_{i'j} \wedge \mathbf{X} \bigwedge_k \psi'_{i'k})$
  - from each disjunct $(\bigwedge_j l'_{i'j} \wedge \mathbf{X} \bigwedge_k \psi'_{i'k})$ generate a new state $S_{ii'}$ (if not already present) and label it as satisfying $\varphi_i \stackrel{\text{def}}{=} \bigwedge_k \psi_{ik}$
- draw an edge from S_i to all states $S_{ii'}$ which satisfy $\bigwedge_k \psi_{ik}$
- (if no next part occurs, $\mathbf{X}\top$ is implicitly assumed, so that an edge to a "true" node is drawn)

On-the-fly Construction of A_φ (Intuition) [cont.]

Step 3: For every state S_i represented by $(\bigwedge_j l_{ij} \wedge \mathbf{X} \overbrace{\bigwedge_k \psi_{ik}}^{\varphi_i})$

- label the incoming edges of S_i with $\bigwedge_j l_{ij}$
- mark that the state S_i satisfies φ
- apply recursively steps 1-2-3 to $\varphi_i \stackrel{\text{def}}{=} \bigwedge_k \psi_{ik}$,
  - rewrite φ_i into $\bigvee_{i'} (\bigwedge_j l'_{i'j} \wedge \mathbf{X} \bigwedge_k \psi'_{i'k})$
  - from each disjunct $(\bigwedge_j l'_{i'j} \wedge \mathbf{X} \bigwedge_k \psi'_{i'k})$ generate a new state $S_{ii'}$ (if not already present) and label it as satisfying $\varphi_i \stackrel{\text{def}}{=} \bigwedge_k \psi_{ik}$
- draw an edge from S_i to all states $S_{ii'}$ which satisfy $\bigwedge_k \psi_{ik}$
- (if no next part occurs, $\mathbf{X}\top$ is implicitly assumed, so that an edge to a “true” node is drawn)

On-the-fly Construction of A_φ (Intuition) [cont.]

Step 3: For every state S_i represented by $(\bigwedge_j l_{ij} \wedge \mathbf{X} \overbrace{\bigwedge_k \psi_{ik}}^{\varphi_i})$

- label the incoming edges of S_i with $\bigwedge_j l_{ij}$
- mark that the state S_i satisfies φ
- apply recursively steps 1-2-3 to $\varphi_i \stackrel{\text{def}}{=} \bigwedge_k \psi_{ik}$,
  - rewrite φ_i into $\bigvee_{i'}(\bigwedge_j l'_{i'j} \wedge \mathbf{X} \bigwedge_k \psi'_{i'k})$
  - from each disjunct $(\bigwedge_j l'_{i'j} \wedge \mathbf{X} \bigwedge_k \psi'_{i'k})$ generate a new state $S_{ii'}$ (if not already present) and label it as satisfying $\varphi_i \stackrel{\text{def}}{=} \bigwedge_k \psi_{ik}$
- draw an edge from S_i to all states $S_{ii'}$ which satisfy $\bigwedge_k \psi_{ik}$
- (if no next part occurs, $\mathbf{X}\top$ is implicitly assumed, so that an edge to a “true” node is drawn)

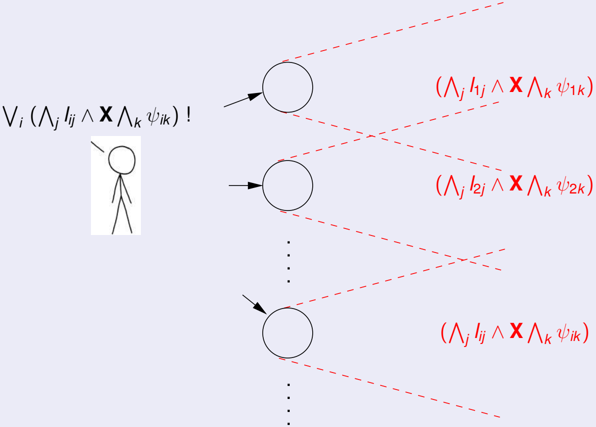
On-the-fly Construction of A_φ (Intuition) [cont.]

φ ??

On-the-fly Construction of A_φ (Intuition) [cont.]

$\bigvee_i (\bigwedge_j l_{ij} \wedge \mathbf{X} \bigwedge_k \psi_{ik})$!

On-the-fly Construction of A_φ (Intuition) [cont.]



$\bigvee_i (\bigwedge_j l_{ij} \wedge \mathbf{X} \bigwedge_k \psi_{ik})$!

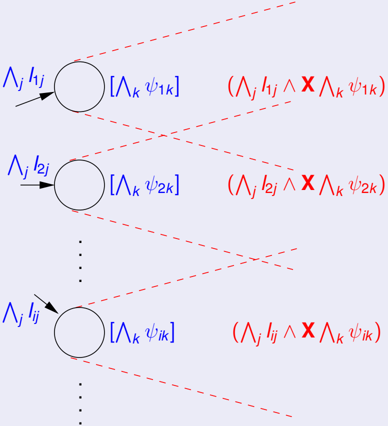
$(\bigwedge_j l_{1j} \wedge \mathbf{X} \bigwedge_k \psi_{1k})$

$(\bigwedge_j l_{2j} \wedge \mathbf{X} \bigwedge_k \psi_{2k})$

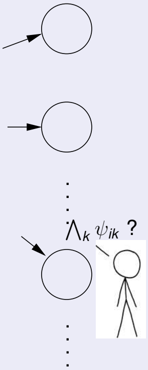
$(\bigwedge_j l_{ij} \wedge \mathbf{X} \bigwedge_k \psi_{ik})$

On-the-fly Construction of A_φ (Intuition) [cont.]

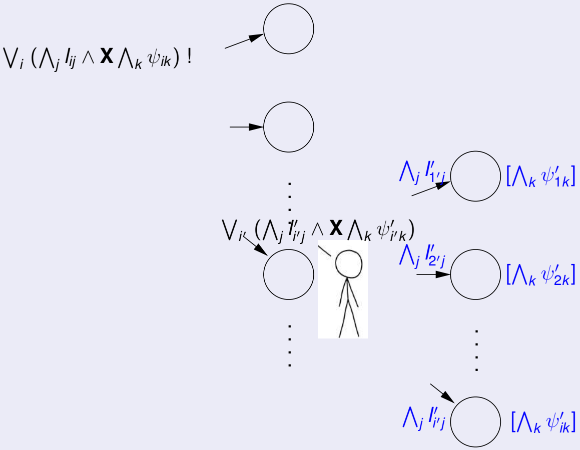
$\bigvee_i (\bigwedge_j l_{ij} \wedge \mathbf{X} \bigwedge_k \psi_{ik})$!



On-the-fly Construction of A_φ (Intuition) [cont.]



On-the-fly Construction of A_φ (Intuition) [cont.]



$\bigvee_i (\bigwedge_j l_{ij} \wedge \mathbf{X} \bigwedge_k \psi_{ik})$!

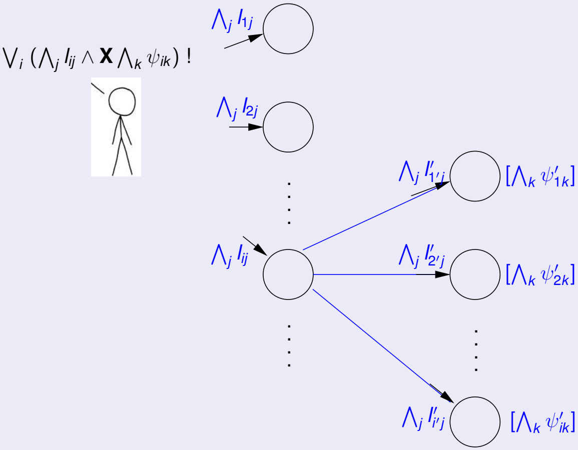
$\bigvee_{i'} (\bigwedge_j l'_{i'j} \wedge \mathbf{X} \bigwedge_k \psi'_{i'k})$

$\bigwedge_j l'_{1'j}$ $[\bigwedge_k \psi'_{1k}]$

$\bigwedge_j l'_{2'j}$ $[\bigwedge_k \psi'_{2k}]$

$\bigwedge_j l'_{i'j}$ $[\bigwedge_k \psi'_{ik}]$

On-the-fly Construction of A_φ (Intuition) [cont.]



On-the-fly Construction of A_φ (Intuition) [cont.]

When the recursive applications of steps 1-3 has terminated and the automata graph has been built, then apply the following:

Step 4: For every $\psi_i \mathbf{U} \varphi_i$, for every state q_j, mark q_j with F_i iff $(\psi_i \mathbf{U} \varphi_i) \notin q_j$ *or* $\varphi_i \in q_j$
(If there is no **U**-subformulas, then mark all states with F_1 —i.e., $FT \overset{\text{def}}{=} \{Q\}$).

Remark

The fact that we initially converted the formula into NNF guarantees that only original positive **U**/**F**-subformulas and negative **R**-/**G**-subformulas are considered in step 4

On-the-fly Construction of A_φ (Intuition) [cont.]

When the recursive applications of steps 1-3 has terminated and the automata graph has been built, then apply the following:

Step 4: For every $\psi_i\mathbf{U}\varphi_i$, for every state q_j, mark q_j with F_i iff $(\psi_i\mathbf{U}\varphi_i) \notin q_j$ *or* $\varphi_i \in q_j$
(If there is no **U**-subformulas, then mark all states with F_1 —i.e., $FT \overset{\text{def}}{=} \{Q\}$).

Remark

The fact that we initially converted the formula into NNF guarantees that only original positive **U**/**F**-subformulas and negative **R**-/**G**-subformulas are considered in step 4

Dealing with $\mathbf{U}$-subformulas: Intuition

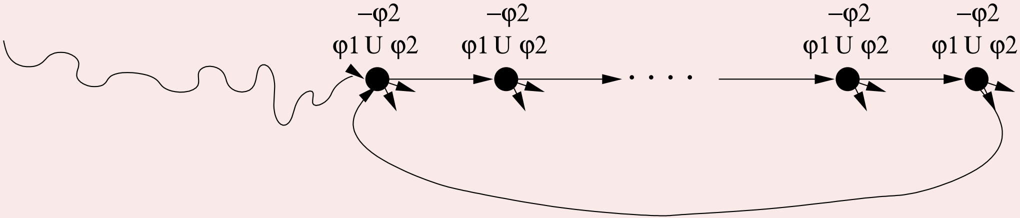
- Tableaux rules: $\varphi_1 \mathbf{U} \varphi_2 \Longleftrightarrow (\varphi_2 \vee (\varphi_1 \wedge \mathbf{X} \varphi_1 \mathbf{U} \varphi_2))$
  are a property, not a definition of $\mathbf{U}$:
  $\Longrightarrow$ they implicitly admit a "weaker" semantics of $\varphi_1 \mathbf{U} \varphi_2$, in which $\varphi_1 \mathbf{U} \varphi_2$ always holds and φ_2 never holds
- It cannot happen that we get into a state s' from which we can enter a path π' in which $\varphi_1 \mathbf{U} \varphi_2$ holds forever and φ_2 never holds.

$\Longrightarrow$ every legal path must touch infinitely often a state where $\neg(\varphi_1 \mathbf{U} \varphi_2) \vee \varphi_2)$ holds

- In LTL: $\neg \mathbf{FG}((\varphi_1 \mathbf{U} \varphi_2) \wedge \neg \varphi_2)$, i.e., $\mathbf{GF}(\neg(\varphi_1 \mathbf{U} \varphi_2) \vee \varphi_2)$ ("avoid bad loops")

Dealing with **U**-subformulas: Intuition

- Tableaux rules: $\varphi_1 \mathbf{U} \varphi_2 \Longleftrightarrow (\varphi_2 \vee (\varphi_1 \wedge \mathbf{X} \varphi_1 \mathbf{U} \varphi_2))$
  are a property, not a definition of **U**:
  $\Longrightarrow$ they implicitly admit a “weaker” semantics of $\varphi_1 \mathbf{U} \varphi_2$, in which $\varphi_1 \mathbf{U} \varphi_2$ always holds and φ_2 never holds
- It cannot happen that we get into a state s' from which we can enter a path π' in which $\varphi_1 \mathbf{U} \varphi_2$ holds forever and φ_2 never holds.

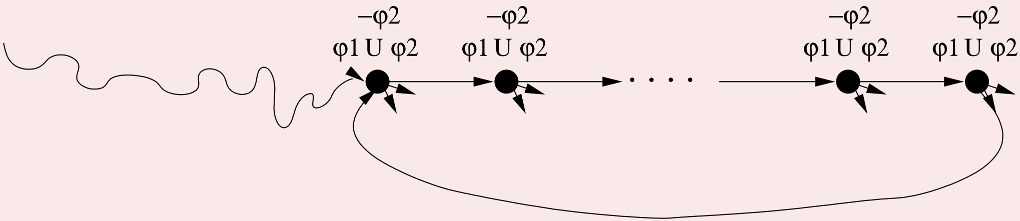


$\Longrightarrow$ every legal path must touch infinitely often a state where $\neg(\varphi_1 \mathbf{U} \varphi_2) \vee \varphi_2)$ holds

- In LTL: $\neg \mathbf{FG}((\varphi_1 \mathbf{U} \varphi_2) \wedge \neg \varphi_2)$, i.e., $\mathbf{GF}(\neg(\varphi_1 \mathbf{U} \varphi_2) \vee \varphi_2)$ (“avoid bad loops”)

Dealing with **U**-subformulas: Intuition

- Tableaux rules: $\varphi_1 \mathbf{U} \varphi_2 \iff (\varphi_2 \vee (\varphi_1 \wedge \mathbf{X} \varphi_1 \mathbf{U} \varphi_2))$ are a property, not a definition of **U**:
  $\Longrightarrow$ they implicitly admit a "weaker" semantics of $\varphi_1 \mathbf{U} \varphi_2$, in which $\varphi_1 \mathbf{U} \varphi_2$ always holds and φ_2 never holds
- It cannot happen that we get into a state s' from which we can enter a path π' in which $\varphi_1 \mathbf{U} \varphi_2$ holds forever and φ_2 never holds.

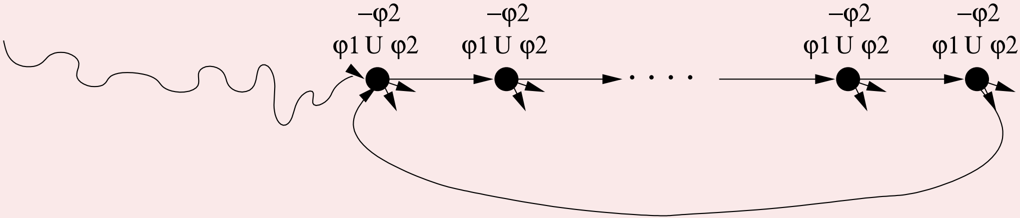


$\Longrightarrow$ every legal path must touch infinitely often a state where $\neg(\varphi_1 \mathbf{U} \varphi_2) \vee \varphi_2)$ holds

- In LTL: $\neg \mathbf{FG}((\varphi_1 \mathbf{U} \varphi_2) \wedge \neg \varphi_2)$, i.e., $\mathbf{GF}(\neg(\varphi_1 \mathbf{U} \varphi_2) \vee \varphi_2)$ ("avoid bad loops")

Dealing with **U**-subformulas: Intuition

- Tableaux rules: $\varphi_1 \mathbf{U} \varphi_2 \Longleftrightarrow (\varphi_2 \vee (\varphi_1 \wedge \mathbf{X} \varphi_1 \mathbf{U} \varphi_2))$ are a property, not a definition of **U**:
  $\Longrightarrow$ they implicitly admit a "weaker" semantics of $\varphi_1 \mathbf{U} \varphi_2$, in which $\varphi_1 \mathbf{U} \varphi_2$ always holds and φ_2 never holds
- It cannot happen that we get into a state s' from which we can enter a path π' in which $\varphi_1 \mathbf{U} \varphi_2$ holds forever and φ_2 never holds.



$\Longrightarrow$ every legal path must touch infinitely often a state where $\neg(\varphi_1 \mathbf{U} \varphi_2) \vee \varphi_2)$ holds

- In LTL: $\neg \mathbf{FG}((\varphi_1 \mathbf{U} \varphi_2) \wedge \neg \varphi_2)$, i.e., $\mathbf{GF}(\neg(\varphi_1 \mathbf{U} \varphi_2) \vee \varphi_2)$ ("avoid bad loops")

On-the-fly Construction of A_φ - State

- Henceforth, a state is represented by a tuple $s := \langle \lambda, \chi, \sigma \rangle$ where:
  - λ is the set of labels
  - χ is the next part, i.e. the set of X-formulas satisfied by s
  - σ is the set of the subformulas of φ satisfied by s (necessary for the fairness definition)
- Given a set of LTL formulas $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$, we define *Cover*$(\Psi) \stackrel{\text{def}}{=}$ *Expand*$(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$ to be the set of initial states of the Buchi automaton representing $\bigwedge_j \psi_j$.
  - *Expand*(Ψ, s) takes as input:
    - a set of LTL formulas $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ to be expanded
    - a state $s \stackrel{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$ under construction

    and returns a set of states $\{\langle \lambda_i, \chi_i, \sigma_i \rangle\}_i$ representing te expansion of Ψ
  - Combines steps 1. and 2. of previous slides

On-the-fly Construction of A_φ - State

- Henceforth, a state is represented by a tuple $s := \langle \lambda, \chi, \sigma \rangle$ where:
  - λ is the set of labels
  - χ is the next part, i.e. the set of X-formulas satisfied by s
  - σ is the set of the subformulas of φ satisfied by s (necessary for the fairness definition)
- Given a set of LTL formulas $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$, we define $\mathit{Cover}(\Psi) \overset{\text{def}}{=} \mathit{Expand}(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$ to be the set of initial states of the Buchi automaton representing $\bigwedge_j \psi_j$.
  - $\mathit{Expand}(\Psi, s)$ takes as input:
    - a set of LTL formulas $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ to be expanded
    - a state $s \overset{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$ under construction

    and returns a set of states $\{\langle \lambda_i, \chi_i, \sigma_i \rangle\}_i$ representing te expansion of Ψ
  - Combines steps 1. and 2. of previous slides

On-the-fly Construction of A_φ - State

- Henceforth, a state is represented by a tuple $s := \langle \lambda, \chi, \sigma \rangle$ where:
  - λ is the set of labels
  - χ is the next part, i.e. the set of X-formulas satisfied by s
  - σ is the set of the subformulas of φ satisfied by s (necessary for the fairness definition)
- Given a set of LTL formulas $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$, we define $\mathit{Cover}(\Psi) \stackrel{\text{def}}{=} \mathit{Expand}(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$ to be the set of initial states of the Buchi automaton representing $\bigwedge_j \psi_j$.
  - $\mathit{Expand}(\Psi, s)$ takes as input:
    - a set of LTL formulas $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ to be expanded
    - a state $s \stackrel{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$ under construction

    and returns a set of states $\{\langle \lambda_i, \chi_i, \sigma_i \rangle\}_i$ representing te expansion of Ψ
  - Combines steps 1. and 2. of previous slides

On-the-fly Construction of A_φ - State

- Henceforth, a state is represented by a tuple $s := \langle \lambda, \chi, \sigma \rangle$ where:
  - λ is the set of labels
  - χ is the next part, i.e. the set of X-formulas satisfied by s
  - σ is the set of the subformulas of φ satisfied by s (necessary for the fairness definition)
- Given a set of LTL formulas $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$, we define $\mathit{Cover}(\Psi) \overset{\text{def}}{=} \mathit{Expand}(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$ to be the set of initial states of the Buchi automaton representing $\bigwedge_j \psi_j$.
  - $\mathit{Expand}(\Psi, s)$ takes as input:
    - a set of LTL formulas $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ to be expanded
    - a state $s \overset{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$ under construction

    and returns a set of states $\{\langle \lambda_i, \chi_i, \sigma_i \rangle\}_i$ representing te expansion of Ψ
  - Combines steps 1. and 2. of previous slides

On-the-fly Construction of A_φ - State

- Henceforth, a state is represented by a tuple $s := \langle \lambda, \chi, \sigma \rangle$ where:
  - λ is the set of labels
  - χ is the next part, i.e. the set of X-formulas satisfied by s
  - σ is the set of the subformulas of φ satisfied by s (necessary for the fairness definition)
- Given a set of LTL formulas $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$, we define $Cover(\Psi) \stackrel{\text{def}}{=} Expand(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$ to be the set of initial states of the Buchi automaton representing $\bigwedge_j \psi_j$.
  - $Expand(\Psi, s)$ takes as input:
    - a set of LTL formulas $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ to be expanded
    - a state $s \stackrel{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$ under construction

    and returns a set of states $\{\langle \lambda_i, \chi_i, \sigma_i \rangle\}_i$ representing te expansion of Ψ
  - Combines steps 1. and 2. of previous slides

On-the-fly Construction of A_φ - State

- Henceforth, a state is represented by a tuple $s := \langle \lambda, \chi, \sigma \rangle$ where:
  - λ is the set of labels
  - χ is the next part, i.e. the set of X-formulas satisfied by s
  - σ is the set of the subformulas of φ satisfied by s (necessary for the fairness definition)
- Given a set of LTL formulas $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$, we define $Cover(\Psi) \overset{\text{def}}{=} Expand(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$ to be the set of initial states of the Buchi automaton representing $\bigwedge_j \psi_j$.
  - $Expand(\Psi, s)$ takes as input:
    - a set of LTL formulas $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ to be expanded
    - a state $s \overset{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$ under construction

    and returns a set of states $\{\langle \lambda_i, \chi_i, \sigma_i \rangle\}_i$ representing te expansion of Ψ
  - Combines steps 1. and 2. of previous slides

On-the-fly Construction of A_φ - State

- Henceforth, a state is represented by a tuple $s := \langle \lambda, \chi, \sigma \rangle$ where:
  - λ is the set of labels
  - χ is the next part, i.e. the set of X-formulas satisfied by s
  - σ is the set of the subformulas of φ satisfied by s (necessary for the fairness definition)
- Given a set of LTL formulas $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$, we define $Cover(\Psi) \overset{\text{def}}{=} Expand(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$ to be the set of initial states of the Buchi automaton representing $\bigwedge_j \psi_j$.
  - *Expand*(Ψ, s) takes as input:
    - a set of LTL formulas $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ to be expanded
    - a state $s \overset{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$ under construction

    and returns a set of states $\{\langle \lambda_i, \chi_i, \sigma_i \rangle\}_i$ representing te expansion of Ψ
  - Combines steps 1. and 2. of previous slides

On-the-fly Construction of A_φ - Expand

Given $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ and $s \stackrel{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$, we define *Expand*$(\Psi, s)$ recursively as follows:

- if $\Psi = \emptyset$, *Expand*$(\Psi, s) = \{s\}$
- if $\bot \in \Psi$, *Expand*$(\Psi, s) = \emptyset$
- if $\top \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  Expand$(\Psi, s) =$ *Expand*$(\Psi \backslash \{\top\}, \langle \lambda, \chi, \sigma \cup \{\top\} \rangle)$
- if $l \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$, l propositional literal
  Expand$(\Psi, s) =$ *Expand*$(\Psi \backslash \{l\}, \langle \lambda \cup \{l\}, \chi, \sigma \cup \{l\} \rangle)$
  (add l to the labels of s and to set of satisfied formulas)
- if $\mathbf{X}\psi \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  Expand$(\Psi, s) =$ *Expand*$(\Psi \backslash \{X\psi\}, \langle \lambda, \chi \cup \{\psi\}, \sigma \cup \{\mathbf{X}\psi\} \rangle)$
  (add ψ to the next part of s and $\mathbf{X}\psi$ to set of satisfied formulas)
- if $\psi_1 \wedge \psi_2 \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  Expand$(\Psi, s) =$ *Expand*$(\Psi \cup \{\psi_1, \psi_2\} \backslash \{\psi_1 \wedge \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \wedge \psi_2\} \rangle)$
  (process both ψ_1 and ψ_2 and add $\psi_1 \wedge \psi_2$ to σ)
- ...

On-the-fly Construction of A_φ - Expand

Given $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ and $s \overset{\text{def}}{=} \langle\lambda, \chi, \sigma\rangle$, we define $Expand(\Psi, s)$ recursively as follows:

- if $\Psi = \emptyset$, $Expand(\Psi, s) = \{s\}$
- if $\bot \in \Psi$, $Expand(\Psi, s) = \emptyset$
- if $\top \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{\top\}, \langle\lambda, \chi, \sigma \cup \{\top\}\rangle)$
- if $l \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$, l propositional literal
  $Expand(\Psi, s) = Expand(\Psi \backslash \{l\}, \langle\lambda \cup \{l\}, \chi, \sigma \cup \{l\}\rangle)$
  (add l to the labels of s and to set of satisfied formulas)
- if $\mathbf{X}\psi \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{X\psi\}, \langle\lambda, \chi \cup \{\psi\}, \sigma \cup \{\mathbf{X}\psi\}\rangle)$
  (add ψ to the next part of s and $\mathbf{X}\psi$ to set of satisfied formulas)
- if $\psi_1 \wedge \psi_2 \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \cup \{\psi_1, \psi_2\} \backslash \{\psi_1 \wedge \psi_2\}, \langle\lambda, \chi, \sigma \cup \{\psi_1 \wedge \psi_2\}\rangle)$
  (process both ψ_1 and ψ_2 and add $\psi_1 \wedge \psi_2$ to σ)
- ...

On-the-fly Construction of A_φ - Expand

Given $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ and $s \stackrel{\text{def}}{=} \langle\lambda, \chi, \sigma\rangle$, we define $Expand(\Psi, s)$ recursively as follows:

- if $\Psi = \emptyset$, $Expand(\Psi, s) = \{s\}$
- if $\bot \in \Psi$, $Expand(\Psi, s) = \emptyset$
- if $\top \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{\top\}, \langle\lambda, \chi, \sigma \cup \{\top\}\rangle)$
- if $l \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$, l propositional literal
  $Expand(\Psi, s) = Expand(\Psi \backslash \{l\}, \langle\lambda \cup \{l\}, \chi, \sigma \cup \{l\}\rangle)$
  (add l to the labels of s and to set of satisfied formulas)
- if $\mathbf{X}\psi \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{X\psi\}, \langle\lambda, \chi \cup \{\psi\}, \sigma \cup \{\mathbf{X}\psi\}\rangle)$
  (add ψ to the next part of s and $\mathbf{X}\psi$ to set of satisfied formulas)
- if $\psi_1 \wedge \psi_2 \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \cup \{\psi_1, \psi_2\} \backslash \{\psi_1 \wedge \psi_2\}, \langle\lambda, \chi, \sigma \cup \{\psi_1 \wedge \psi_2\}\rangle)$
  (process both ψ_1 and ψ_2 and add $\psi_1 \wedge \psi_2$ to σ)
- ...

On-the-fly Construction of A_φ - Expand

Given $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ and $s \stackrel{\text{def}}{=} \langle\lambda, \chi, \sigma\rangle$, we define $Expand(\Psi, s)$ recursively as follows:

- if $\Psi = \emptyset$, $Expand(\Psi, s) = \{s\}$
- if $\bot \in \Psi$, $Expand(\Psi, s) = \emptyset$
- if $\top \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{\top\}, \langle\lambda, \chi, \sigma \cup \{\top\}\rangle)$
- if $l \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$, l propositional literal
  $Expand(\Psi, s) = Expand(\Psi \backslash \{l\}, \langle\lambda \cup \{l\}, \chi, \sigma \cup \{l\}\rangle)$
  (add l to the labels of s and to set of satisfied formulas)
- if $\mathbf{X}\psi \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{X\psi\}, \langle\lambda, \chi \cup \{\psi\}, \sigma \cup \{\mathbf{X}\psi\}\rangle)$
  (add ψ to the next part of s and $\mathbf{X}\psi$ to set of satisfied formulas)
- if $\psi_1 \wedge \psi_2 \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \cup \{\psi_1, \psi_2\} \backslash \{\psi_1 \wedge \psi_2\}, \langle\lambda, \chi, \sigma \cup \{\psi_1 \wedge \psi_2\}\rangle)$
  (process both ψ_1 and ψ_2 and add $\psi_1 \wedge \psi_2$ to σ)
- ...

On-the-fly Construction of A_φ - Expand

Given $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ and $s \overset{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$, we define $Expand(\Psi, s)$ recursively as follows:

- if $\Psi = \emptyset$, $Expand(\Psi, s) = \{s\}$
- if $\bot \in \Psi$, $Expand(\Psi, s) = \emptyset$
- if $\top \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{\top\}, \langle \lambda, \chi, \sigma \cup \{\top\} \rangle)$
- if $l \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$, l propositional literal
  $Expand(\Psi, s) = Expand(\Psi \backslash \{l\}, \langle \lambda \cup \{l\}, \chi, \sigma \cup \{l\} \rangle)$
  (add l to the labels of s and to set of satisfied formulas)
- if $\mathbf{X}\psi \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{\mathbf{X}\psi\}, \langle \lambda, \chi \cup \{\psi\}, \sigma \cup \{\mathbf{X}\psi\} \rangle)$
  (add ψ to the next part of s and $\mathbf{X}\psi$ to set of satisfied formulas)
- if $\psi_1 \wedge \psi_2 \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $Expand(\Psi, s) = Expand(\Psi \cup \{\psi_1, \psi_2\} \backslash \{\psi_1 \wedge \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \wedge \psi_2\} \rangle)$
  (process both ψ_1 and ψ_2 and add $\psi_1 \wedge \psi_2$ to σ)
- ...

On-the-fly Construction of A_φ - Expand

Given $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ and $s \overset{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$, we define $Expand(\Psi, s)$ recursively as follows:

- if $\Psi = \emptyset$, $Expand(\Psi, s) = \{s\}$
- if $\bot \in \Psi$, $Expand(\Psi, s) = \emptyset$
- if $\top \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{\top\}, \langle \lambda, \chi, \sigma \cup \{\top\} \rangle)$
- if $l \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$, l propositional literal
  $Expand(\Psi, s) = Expand(\Psi \backslash \{l\}, \langle \lambda \cup \{l\}, \chi, \sigma \cup \{l\} \rangle)$
  (add l to the labels of s and to set of satisfied formulas)
- if $\mathbf{X}\psi \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{X\psi\}, \langle \lambda, \chi \cup \{\psi\}, \sigma \cup \{\mathbf{X}\psi\} \rangle)$
  (add ψ to the next part of s and $\mathbf{X}\psi$ to set of satisfied formulas)
- if $\psi_1 \wedge \psi_2 \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $Expand(\Psi, s) = Expand(\Psi \cup \{\psi_1, \psi_2\} \backslash \{\psi_1 \wedge \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \wedge \psi_2\} \rangle)$
  (process both ψ_1 and ψ_2 and add $\psi_1 \wedge \psi_2$ to σ)
- ...

On-the-fly Construction of A_φ - Expand

Given $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ and $s \stackrel{\text{def}}{=} \langle\lambda, \chi, \sigma\rangle$, we define $Expand(\Psi, s)$ recursively as follows:

- if $\Psi = \emptyset$, $Expand(\Psi, s) = \{s\}$
- if $\bot \in \Psi$, $Expand(\Psi, s) = \emptyset$
- if $\top \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{\top\}, \langle\lambda, \chi, \sigma \cup \{\top\}\rangle)$
- if $l \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$, l propositional literal
  $Expand(\Psi, s) = Expand(\Psi \backslash \{l\}, \langle\lambda \cup \{l\}, \chi, \sigma \cup \{l\}\rangle)$
  (add l to the labels of s and to set of satisfied formulas)
- if $\mathbf{X}\psi \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \backslash \{X\psi\}, \langle\lambda, \chi \cup \{\psi\}, \sigma \cup \{\mathbf{X}\psi\}\rangle)$
  (add ψ to the next part of s and $\mathbf{X}\psi$ to set of satisfied formulas)
- if $\psi_1 \wedge \psi_2 \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $Expand(\Psi, s) = Expand(\Psi \cup \{\psi_1, \psi_2\} \backslash \{\psi_1 \wedge \psi_2\}, \langle\lambda, \chi, \sigma \cup \{\psi_1 \wedge \psi_2\}\rangle)$
  (process both ψ_1 and ψ_2 and add $\psi_1 \wedge \psi_2$ to σ)
- ...

On-the-fly Construction of A_φ - Expand

Given $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ and $s \overset{\text{def}}{=} \langle\lambda, \chi, \sigma\rangle$, we define $\mathit{Expand}(\Psi, s)$ recursively as follows:

- ...
- if $\psi_1 \vee \psi_2 \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $$\begin{aligned}\mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi \cup \{\psi_1\} \backslash \{\psi_1 \vee \psi_2\}, \langle\lambda, \chi, \sigma \cup \{\psi_1 \vee \psi_2\}\rangle) \\ \cup {} & \mathit{Expand}(\Psi \cup \{\psi_2\} \backslash \{\psi_1 \vee \psi_2\}, \langle\lambda, \chi, \sigma \cup \{\psi_1 \vee \psi_2\}\rangle)\end{aligned}$$
  (split *s* into two copies, process ψ_2 on the first, ψ_1 on the second, add $\psi_1 \vee \psi_2$ to σ)
- if $\psi_1 \mathbf{U} \psi_2 \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $$\begin{aligned}\mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi \cup \{\psi_1\} \backslash \{\psi_1 \mathbf{U} \psi_2\}, \langle\lambda, \chi \cup \{\psi_1 \mathbf{U} \psi_2\}, \sigma \cup \{\psi_1 \mathbf{U} \psi_2\}\rangle) \\ \cup {} & \mathit{Expand}(\Psi \cup \{\psi_2\} \backslash \{\psi_1 \mathbf{U} \psi_2\}, \langle\lambda, \chi, \sigma \cup \{\psi_1 \mathbf{U} \psi_2\}\rangle)\end{aligned}$$
  (split *s* into two copies and process ψ_1 on the first, ψ_2 on the second, add $\psi_1 \mathbf{U} \psi_2$ to σ)
- if $\psi_1 \mathbf{R} \psi_2 \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $$\begin{aligned}\mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi \cup \{\psi_2\} \backslash \{\psi_1 \mathbf{R} \psi_2\}, \langle\lambda, \chi \cup \{\psi_1 \mathbf{R} \psi_2\}, \sigma \cup \{\psi_1 \mathbf{R} \psi_2\}\rangle) \\ \cup {} & \mathit{Expand}(\Psi \cup \{\psi_1, \psi_2\} \backslash \{\psi_1 \mathbf{R} \psi_2\}, \langle\lambda, \chi, \sigma \cup \{\psi_1 \mathbf{R} \psi_2\}\rangle)\end{aligned}$$
  (split *s* into two copies and process ψ_1 on the first, ψ_2 on the second, add $\psi_1 \mathbf{R} \psi_2$ to σ)

On-the-fly Construction of A_φ - Expand

Given $\Psi \stackrel{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ and $s \stackrel{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$, we define $\mathit{Expand}(\Psi, s)$ recursively as follows:

- ...
- if $\psi_1 \vee \psi_2 \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $$\begin{aligned} \mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi \cup \{\psi_1\} \backslash \{\psi_1 \vee \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \vee \psi_2\} \rangle) \\ \cup {} & \mathit{Expand}(\Psi \cup \{\psi_2\} \backslash \{\psi_1 \vee \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \vee \psi_2\} \rangle) \end{aligned}$$
  (split s into two copies, process ψ_2 on the first, ψ_1 on the second, add $\psi_1 \vee \psi_2$ to σ)
- if $\psi_1 \mathbf{U} \psi_2 \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $$\begin{aligned} \mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi \cup \{\psi_1\} \backslash \{\psi_1 \mathbf{U} \psi_2\}, \langle \lambda, \chi \cup \{\psi_1 \mathbf{U} \psi_2\}, \sigma \cup \{\psi_1 \mathbf{U} \psi_2\} \rangle) \\ \cup {} & \mathit{Expand}(\Psi \cup \{\psi_2\} \backslash \{\psi_1 \mathbf{U} \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \mathbf{U} \psi_2\} \rangle) \end{aligned}$$
  (split s into two copies and process ψ_1 on the first, ψ_2 on the second, add $\psi_1 \mathbf{U} \psi_2$ to σ)
- if $\psi_1 \mathbf{R} \psi_2 \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $$\begin{aligned} \mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi \cup \{\psi_2\} \backslash \{\psi_1 \mathbf{R} \psi_2\}, \langle \lambda, \chi \cup \{\psi_1 \mathbf{R} \psi_2\}, \sigma \cup \{\psi_1 \mathbf{R} \psi_2\} \rangle) \\ \cup {} & \mathit{Expand}(\Psi \cup \{\psi_1, \psi_2\} \backslash \{\psi_1 \mathbf{R} \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \mathbf{R} \psi_2\} \rangle) \end{aligned}$$
  (split s into two copies and process ψ_1 on the first, ψ_2 on the second, add $\psi_1 \mathbf{R} \psi_2$ to σ)

On-the-fly Construction of A_φ - Expand

Given $\Psi \overset{\text{def}}{=} \{\psi_1, ..., \psi_k\}$ and $s \overset{\text{def}}{=} \langle \lambda, \chi, \sigma \rangle$, we define $\mathit{Expand}(\Psi, s)$ recursively as follows:

- ...
- if $\psi_1 \vee \psi_2 \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $$\begin{aligned} \mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi \cup \{\psi_1\} \setminus \{\psi_1 \vee \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \vee \psi_2\} \rangle) \\ \cup {} & \mathit{Expand}(\Psi \cup \{\psi_2\} \setminus \{\psi_1 \vee \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \vee \psi_2\} \rangle) \end{aligned}$$
  (split s into two copies, process ψ_2 on the first, ψ_1 on the second, add $\psi_1 \vee \psi_2$ to σ)
- if $\psi_1 \mathbf{U} \psi_2 \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $$\begin{aligned} \mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi \cup \{\psi_1\} \setminus \{\psi_1 \mathbf{U} \psi_2\}, \langle \lambda, \chi \cup \{\psi_1 \mathbf{U} \psi_2\}, \sigma \cup \{\psi_1 \mathbf{U} \psi_2\} \rangle) \\ \cup {} & \mathit{Expand}(\Psi \cup \{\psi_2\} \setminus \{\psi_1 \mathbf{U} \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \mathbf{U} \psi_2\} \rangle) \end{aligned}$$
  (split s into two copies and process ψ_1 on the first, ψ_2 on the second, add $\psi_1 \mathbf{U} \psi_2$ to σ)
- if $\psi_1 \mathbf{R} \psi_2 \in \Psi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $$\begin{aligned} \mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi \cup \{\psi_2\} \setminus \{\psi_1 \mathbf{R} \psi_2\}, \langle \lambda, \chi \cup \{\psi_1 \mathbf{R} \psi_2\}, \sigma \cup \{\psi_1 \mathbf{R} \psi_2\} \rangle) \\ \cup {} & \mathit{Expand}(\Psi \cup \{\psi_1, \psi_2\} \setminus \{\psi_1 \mathbf{R} \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \mathbf{R} \psi_2\} \rangle) \end{aligned}$$
  (split s into two copies and process ψ_1 on the first, ψ_2 on the second, add $\psi_1 \mathbf{R} \psi_2$ to σ)

On-the-fly Construction of A_φ - Expand

Two relevant subcases: $\mathbf{F}\psi \overset{\text{def}}{=} \top\mathbf{U}\psi$ and $\mathbf{G}\psi \overset{\text{def}}{=} \bot\mathbf{R}\psi$

- if $\mathbf{F}\psi \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $$\begin{aligned} \mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi\backslash\{\mathbf{F}\psi\}, \langle\lambda, \chi \cup \{\mathbf{F}\psi\}, \sigma \cup \{\mathbf{F}\psi\}\rangle) \\ & \cup\ \mathit{Expand}(\Psi \cup \{\psi\}\backslash\{\mathbf{F}\psi\}, \langle\lambda, \chi, \sigma \cup \{\mathbf{F}\psi\}\rangle) \end{aligned}$$
- if $\mathbf{G}\psi \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $$\mathit{Expand}(\Psi, s) = \mathit{Expand}(\Psi \cup \{\psi\}\backslash\{\mathbf{G}\psi\}, \langle\lambda, \chi \cup \{\mathbf{G}\psi\}, \sigma \cup \{\mathbf{G}\psi\}\rangle)$$
  (Note: $\mathit{Expand}(\Psi \cup \{\bot, \psi\}\backslash\{\mathbf{G}\psi\}, \ldots) = \emptyset$.)

On-the-fly Construction of A_φ - Expand

Two relevant subcases: $\mathbf{F}\psi \overset{\text{def}}{=} \top\mathbf{U}\psi$ and $\mathbf{G}\psi \overset{\text{def}}{=} \bot\mathbf{R}\psi$

- if $\mathbf{F}\psi \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,

  $$\begin{aligned} \mathit{Expand}(\Psi, s) = {} & \mathit{Expand}(\Psi \backslash \{\mathbf{F}\psi\}, \langle\lambda, \chi \cup \{\mathbf{F}\psi\}, \sigma \cup \{\mathbf{F}\psi\}\rangle) \\ & \cup\ \mathit{Expand}(\Psi \cup \{\psi\} \backslash \{\mathbf{F}\psi\}, \langle\lambda, \chi, \sigma \cup \{\mathbf{F}\psi\}\rangle) \end{aligned}$$

- if $\mathbf{G}\psi \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,

  $$\mathit{Expand}(\Psi, s) = \mathit{Expand}(\Psi \cup \{\psi\} \backslash \{\mathbf{G}\psi\}, \langle\lambda, \chi \cup \{\mathbf{G}\psi\}, \sigma \cup \{\mathbf{G}\psi\}\rangle)$$

  (Note: $\mathit{Expand}(\Psi \cup \{\bot, \psi\} \backslash \{\mathbf{G}\psi\}, \ldots) = \emptyset$.)

On-the-fly Construction of A_φ - Expand

Two relevant subcases: $\mathbf{F}\psi \overset{\text{def}}{=} \top\mathbf{U}\psi$ and $\mathbf{G}\psi \overset{\text{def}}{=} \bot\mathbf{R}\psi$

- if $\mathbf{F}\psi \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $$Expand(\Psi, s) = Expand(\Psi \backslash \{\mathbf{F}\psi\}, \langle\lambda, \chi \cup \{\mathbf{F}\psi\}, \sigma \cup \{\mathbf{F}\psi\}\rangle) \\ \cup Expand(\Psi \cup \{\psi\} \backslash \{\mathbf{F}\psi\}, \langle\lambda, \chi, \sigma \cup \{\mathbf{F}\psi\}\rangle)$$
- if $\mathbf{G}\psi \in \Psi$ and $s = \langle\lambda, \chi, \sigma\rangle$,
  $$Expand(\Psi, s) = Expand(\Psi \cup \{\psi\} \backslash \{\mathbf{G}\psi\}, \langle\lambda, \chi \cup \{\mathbf{G}\psi\}, \sigma \cup \{\mathbf{G}\psi\}\rangle)$$
  (Note: $Expand(\Psi \cup \{\bot, \psi\} \backslash \{\mathbf{G}\psi\}, \ldots) = \emptyset$.)

Definition of A_φ

Given a set of LTL formulas Ψ, we define $\mathit{Cover}(\Psi) \overset{\text{def}}{=} \mathit{Expand}(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$.
For an LTL formula φ, we construct a Generalized NBA $A_\varphi = (Q, \Sigma, \delta, I, FT)$ as follows:

- $\Sigma \;=\; 3^{vars(\varphi)}$ ($v \in \{\top, \bot, *\}$, "$*$" is "don't care")
- Q is the smallest set such that
  - $\mathit{Cover}(\{\varphi\}) \subseteq Q$
  - if $\langle \lambda, \chi, \sigma \rangle \in Q$, then $\mathit{Cover}(\chi) \in Q$
- $Q_0 = \mathit{Cover}(\{\varphi\})$.
- $s \xrightarrow{\lambda'} s' \in \delta$ iff, $s = \langle \lambda, \chi, \sigma \rangle$, $s' = \langle \lambda', \chi', \sigma' \rangle$ and $s' \in \mathit{Cover}(\chi)$
- $FT = \langle F_1, F_2, ..., F_k \rangle$ where, for all $(\psi_i \mathbf{U} \varphi_i)$ occurring positively in φ,
  $F_i \;=\; \{\langle \lambda, \chi, \sigma \rangle \in Q \mid (\psi_i \mathbf{U} \varphi_i) \notin \sigma \;\; \mathit{or} \;\; \varphi_i \in \sigma\}$.
  (If there is no **U**-subformulas, then $FT \overset{\text{def}}{=} \{Q\}$).

Definition of A_φ

Given a set of LTL formulas Ψ, we define $\mathit{Cover}(\Psi) \stackrel{\text{def}}{=} \mathit{Expand}(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$.
For an LTL formula φ, we construct a Generalized NBA $A_\varphi = (Q, \Sigma, \delta, I, FT)$ as follows:

- $\Sigma = 3^{vars(\varphi)}$ ($v \in \{\top, \bot, *\}$, "$*$" is "don't care")
- Q is the smallest set such that
  - $\mathit{Cover}(\{\varphi\}) \subseteq Q$
  - if $\langle \lambda, \chi, \sigma \rangle \in Q$, then $\mathit{Cover}(\chi) \in Q$
- $Q_0 = \mathit{Cover}(\{\varphi\})$.
- $s \xrightarrow{\lambda'} s' \in \delta$ iff, $s = \langle \lambda, \chi, \sigma \rangle$, $s' = \langle \lambda', \chi', \sigma' \rangle$ and $s' \in \mathit{Cover}(\chi)$
- $FT = \langle F_1, F_2, ..., F_k \rangle$ where, for all $(\psi_i \mathbf{U} \varphi_i)$ occurring positively in φ,
  $F_i = \{ \langle \lambda, \chi, \sigma \rangle \in Q \mid (\psi_i \mathbf{U} \varphi_i) \notin \sigma \ \mathit{or} \ \varphi_i \in \sigma \}$.
  (If there is no **U**-subformulas, then $FT \stackrel{\text{def}}{=} \{Q\}$).

Definition of A_φ

Given a set of LTL formulas Ψ, we define $Cover(\Psi) \stackrel{\text{def}}{=} Expand(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$.
For an LTL formula φ, we construct a Generalized NBA $A_\varphi = (Q, \Sigma, \delta, I, FT)$ as follows:

- $\Sigma = 3^{vars(\varphi)}$ ($v \in \{\top, \bot, *\}$, "$*$" is "don't care")
- Q is the smallest set such that
  - $Cover(\{\varphi\}) \subseteq Q$
  - if $\langle \lambda, \chi, \sigma \rangle \in Q$, then $Cover(\chi) \in Q$
- $Q_0 = Cover(\{\varphi\})$.
- $s \xrightarrow{\lambda'} s' \in \delta$ iff, $s = \langle \lambda, \chi, \sigma \rangle$, $s' = \langle \lambda', \chi', \sigma' \rangle$ and $s' \in Cover(\chi)$
- $FT = \langle F_1, F_2, ..., F_k \rangle$ where, for all $(\psi_i \mathbf{U} \varphi_i)$ occurring positively in φ,
  $F_i = \{\langle \lambda, \chi, \sigma \rangle \in Q \mid (\psi_i \mathbf{U} \varphi_i) \notin \sigma \ \ or \ \ \varphi_i \in \sigma\}$.
  (If there is no **U**-subformulas, then $FT \stackrel{\text{def}}{=} \{Q\}$).

Definition of A_φ

Given a set of LTL formulas Ψ, we define $Cover(\Psi) \overset{\text{def}}{=} Expand(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$.
For an LTL formula φ, we construct a Generalized NBA $A_\varphi = (Q, \Sigma, \delta, I, FT)$ as follows:

- $\Sigma = 3^{vars(\varphi)}$ ($v \in \{\top, \bot, *\}$, "$*$" is "don't care")
- Q is the smallest set such that
  - $Cover(\{\varphi\}) \subseteq Q$
  - if $\langle \lambda, \chi, \sigma \rangle \in Q$, then $Cover(\chi) \in Q$
- $Q_0 = Cover(\{\varphi\})$.
- $s \xrightarrow{\lambda'} s' \in \delta$ iff, $s = \langle \lambda, \chi, \sigma \rangle$, $s' = \langle \lambda', \chi', \sigma' \rangle$ and $s' \in Cover(\chi)$
- $FT = \langle F_1, F_2, \ldots, F_k \rangle$ where, for all $(\psi_i \mathbf{U} \varphi_i)$ occurring positively in φ,
  $F_i = \{\langle \lambda, \chi, \sigma \rangle \in Q \mid (\psi_i \mathbf{U} \varphi_i) \notin \sigma \textit{ or } \varphi_i \in \sigma\}$.
  (If there is no **U**-subformulas, then $FT \overset{\text{def}}{=} \{Q\}$).

Definition of A_φ

Given a set of LTL formulas Ψ, we define $Cover(\Psi) \overset{\text{def}}{=} Expand(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$.
For an LTL formula φ, we construct a Generalized NBA $A_\varphi = (Q, \Sigma, \delta, I, FT)$ as follows:

- $\Sigma = 3^{vars(\varphi)}$ ($v \in \{\top, \bot, *\}$, "$*$" is "don't care")
- Q is the smallest set such that
  - $Cover(\{\varphi\}) \subseteq Q$
  - if $\langle \lambda, \chi, \sigma \rangle \in Q$, then $Cover(\chi) \in Q$
- $Q_0 = Cover(\{\varphi\})$.
- $s \xrightarrow{\lambda'} s' \in \delta$ iff, $s = \langle \lambda, \chi, \sigma \rangle$, $s' = \langle \lambda', \chi', \sigma' \rangle$ and $s' \in Cover(\chi)$
- $FT = \langle F_1, F_2, ..., F_k \rangle$ where, for all $(\psi_i \mathbf{U} \varphi_i)$ occurring positively in φ, $F_i = \{\langle \lambda, \chi, \sigma \rangle \in Q \mid (\psi_i \mathbf{U} \varphi_i) \notin \sigma \ \ or \ \ \varphi_i \in \sigma\}$.
  (If there is no **U**-subformulas, then $FT \overset{\text{def}}{=} \{Q\}$).

Definition of A_φ

Given a set of LTL formulas Ψ, we define $Cover(\Psi) \overset{\text{def}}{=} Expand(\Psi, \langle\emptyset, \emptyset, \emptyset\rangle)$.
For an LTL formula φ, we construct a Generalized NBA $A_\varphi = (Q, \Sigma, \delta, I, FT)$ as follows:

- $\Sigma = 3^{vars(\varphi)}$ ($v \in \{\top, \bot, *\}$, "$*$" is "don't care")
- Q is the smallest set such that
  - $Cover(\{\varphi\}) \subseteq Q$
  - if $\langle\lambda, \chi, \sigma\rangle \in Q$, then $Cover(\chi) \in Q$
- $Q_0 = Cover(\{\varphi\})$.
- $s \xrightarrow{\lambda'} s' \in \delta$ iff, $s = \langle\lambda, \chi, \sigma\rangle$, $s' = \langle\lambda', \chi', \sigma'\rangle$ and $s' \in Cover(\chi)$
- $FT = \langle F_1, F_2, ..., F_k\rangle$ where, for all $(\psi_i \mathbf{U} \varphi_i)$ occurring positively in φ,
  $F_i = \{\langle\lambda, \chi, \sigma\rangle \in Q \mid (\psi_i \mathbf{U} \varphi_i) \notin \sigma \text{ or } \varphi_i \in \sigma\}$.
  (If there is no **U**-subformulas, then $FT \overset{\text{def}}{=} \{Q\}$).

Definition of A_φ

Given a set of LTL formulas Ψ, we define $Cover(\Psi) \overset{\text{def}}{=} Expand(\Psi, \langle\emptyset, \emptyset, \emptyset\rangle)$.
For an LTL formula φ, we construct a Generalized NBA $A_\varphi = (Q, \Sigma, \delta, I, FT)$ as follows:

- $\Sigma = 3^{vars(\varphi)}$ ($v \in \{\top, \bot, *\}$, "$*$" is "don't care")
- Q is the smallest set such that
  - $Cover(\{\varphi\}) \subseteq Q$
  - if $\langle\lambda, \chi, \sigma\rangle \in Q$, then $Cover(\chi) \in Q$
- $Q_0 = Cover(\{\varphi\})$.
- $s \xrightarrow{\lambda'} s' \in \delta$ iff, $s = \langle\lambda, \chi, \sigma\rangle$, $s' = \langle\lambda', \chi', \sigma'\rangle$ and $s' \in Cover(\chi)$
- $FT = \langle F_1, F_2, ..., F_k\rangle$ where, for all $(\psi_i \mathbf{U} \varphi_i)$ occurring positively in φ,
  $F_i = \{\langle\lambda, \chi, \sigma\rangle \in Q \mid (\psi_i \mathbf{U} \varphi_i) \notin \sigma \;\; or \;\; \varphi_i \in \sigma\}$.
  (If there is no **U**-subformulas, then $FT \overset{\text{def}}{=} \{Q\}$).

Definition of A_φ

Given a set of LTL formulas Ψ, we define $\mathit{Cover}(\Psi) \stackrel{\text{def}}{=} \mathit{Expand}(\Psi, \langle \emptyset, \emptyset, \emptyset \rangle)$.
For an LTL formula φ, we construct a Generalized NBA $A_\varphi = (Q, \Sigma, \delta, I, FT)$ as follows:

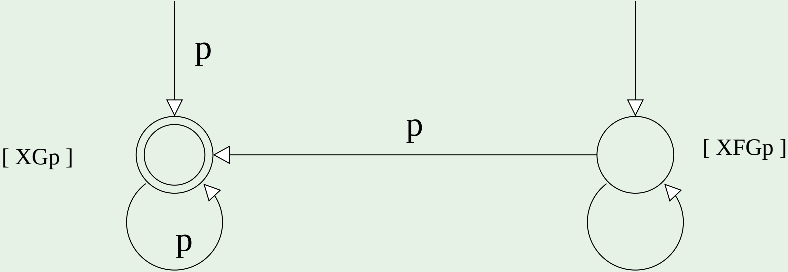
- $\Sigma \; = \; 3^{vars(\varphi)}$ ($v \in \{\top, \bot, *\}$, "$*$" is "don't care")
- Q is the smallest set such that
  - $\mathit{Cover}(\{\varphi\}) \subseteq Q$
  - if $\langle \lambda, \chi, \sigma \rangle \in Q$, then $\mathit{Cover}(\chi) \in Q$
- $Q_0 = \mathit{Cover}(\{\varphi\})$.
- $s \xrightarrow{\lambda'} s' \in \delta$ iff, $s = \langle \lambda, \chi, \sigma \rangle$, $s' = \langle \lambda', \chi', \sigma' \rangle$ and $s' \in \mathit{Cover}(\chi)$
- $FT = \langle F_1, F_2, ..., F_k \rangle$ where, for all $(\psi_i \mathbf{U} \varphi_i)$ occurring positively in φ,
  $F_i \; = \; \{\langle \lambda, \chi, \sigma \rangle \in Q \mid (\psi_i \mathbf{U} \varphi_i) \notin \sigma \;\; \mathit{or} \;\; \varphi_i \in \sigma\}$.
  (If there is no **U**-subformulas, then $FT \stackrel{\text{def}}{=} \{Q\}$).

Example: $\varphi = \mathbf{FG}p$

- $$\begin{aligned}&\mathit{Cover}(\{\mathbf{FG}p\})\\ &= \mathit{Expand}(\{\mathbf{FG}p\}, \langle\emptyset,\emptyset,\emptyset\rangle)\\ &= \mathit{Expand}(\emptyset, \langle\emptyset, \{\mathbf{FG}p\}, \{\mathbf{FG}p\}\rangle) \cup \mathit{Expand}(\{\mathbf{G}p\}, \langle\emptyset,\emptyset,\{\mathbf{FG}p\}\rangle)\\ &= \{\langle\emptyset, \{\mathbf{FG}p\}, \{\mathbf{FG}p\}\rangle\} \cup \mathit{Expand}(\{p\}, \langle\emptyset, \{\mathbf{G}p\}, \{\mathbf{FG}p, \mathbf{G}p\}\rangle)\\ &= \{\langle\emptyset, \{\mathbf{FG}p\}, \{\mathbf{FG}p\}\rangle\} \cup \mathit{Expand}(\emptyset, \langle\{p\}, \{\mathbf{G}p\}, \{\mathbf{FG}p, \mathbf{G}p, p\}\rangle)\\ &= \{\langle\emptyset, \{\mathbf{FG}p\}, \{\mathbf{FG}p\}\rangle, \langle\{p\}, \{\mathbf{G}p\}, \{\mathbf{FG}p, \mathbf{G}p, p\}\rangle\}\end{aligned}$$
- $$\begin{aligned}\mathit{Cover}(\{\mathbf{G}p\}) &= \mathit{Expand}(\{\mathbf{G}p\}, \langle\emptyset,\emptyset,\emptyset\rangle)\\ &= \mathit{Expand}(\{p\}, \langle\emptyset, \{\mathbf{G}p\}, \{\mathbf{G}p\}\rangle)\\ &= \mathit{Expand}(\emptyset, \langle\{p\}, \{\mathbf{G}p\}, \{\mathbf{G}p, p\}\rangle)\\ &= \{\langle\{p\}, \{\mathbf{G}p\}, \{\mathbf{G}p, p\}\rangle\}\end{aligned}$$
- Optimization:
  merge $\langle\{p\}, \{\mathbf{G}p\}, \{\mathbf{FG}p, \mathbf{G}p, p\}\rangle$ and $\langle\{p\}, \{\mathbf{G}p\}, \{\mathbf{G}p, p\}\rangle$

Example: $\varphi = \mathbf{FG}p$

- Call $s_1 = \langle \emptyset, \{\mathbf{FG}p\}, \{\mathbf{FG}p\} \rangle$, $s_2 = \langle \{p\}, \{\mathbf{G}p\}, \{\mathbf{FG}p, \mathbf{G}p, p\} \rangle$
- $Q = \{s_1, s_2\}$
- $Q_0 = \{s_1, s_2\}$.
- $T: \quad s_1 \rightarrow \{s_1, s_2\},$
  $ s_2 \rightarrow \{s_2\}$
- $FT = \langle F_1 \rangle$ where $F_1 = \{s_2\}$.

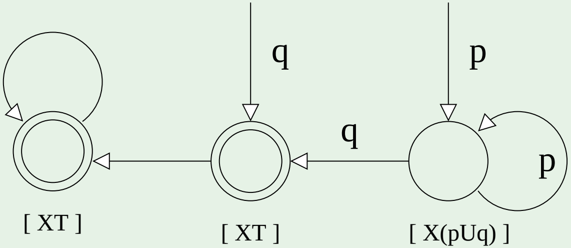


Example: $\varphi = p\mathbf{U}q$

- $$\begin{aligned} & \mathit{Cover}(\{p\mathbf{U}q\}) \\ & = \mathit{Expand}(\{p\mathbf{U}q\}, \langle\emptyset, \emptyset, \emptyset\rangle) \\ & = \mathit{Expand}(\{p\}, \langle\emptyset, \{p\mathbf{U}q\}, \{p\mathbf{U}q\}\rangle) \cup \mathit{Expand}(\{q\}, \langle\emptyset, \emptyset, \{p\mathbf{U}q\}\rangle) \\ & = \mathit{Expand}(\emptyset, \langle\{p\}, \{p\mathbf{U}q\}, \{p\mathbf{U}q, p\}\rangle) \cup \mathit{Expand}(\emptyset, \langle\{q\}, \emptyset, \{p\mathbf{U}q, q\}\rangle) \\ & = \{\langle\{p\}, \{p\mathbf{U}q\}, \{p\mathbf{U}q, p\}\rangle\} \cup \{\langle\{q\}, \{\top\}, \{p\mathbf{U}q, q\}\rangle\} \end{aligned}$$
- $\mathit{Cover}(\{\top\}) = \quad \{\langle\emptyset, \{\top\}, \{\top\}\rangle\}$

Example: $\varphi = p\mathbf{U}q$

- Let $s_1 =_{def} \langle \{p\}, \{p\mathbf{U}q\}, \{p\mathbf{U}q, p\} \rangle$, $s_2 =_{def} \langle \{q\}, \{\top\}, \{p\mathbf{U}q, q\} \rangle$, $s_3 =_{def} \langle \emptyset, \{\top\}, \{\top\} \rangle$.
- $Q = \{s_1, s_2, s_3\}$,
- $Q_0 = \{s_1, s_2\}$,
- $$\begin{aligned} T: \quad & s_1 \rightarrow \{s_1, s_2\}, \\ & s_2 \rightarrow \{s_3\} \\ & s_3 \rightarrow \{s_3\} \end{aligned}$$
- $FT = \langle F_1 \rangle$ where $F_1 = \{s_2, s_3\}$.



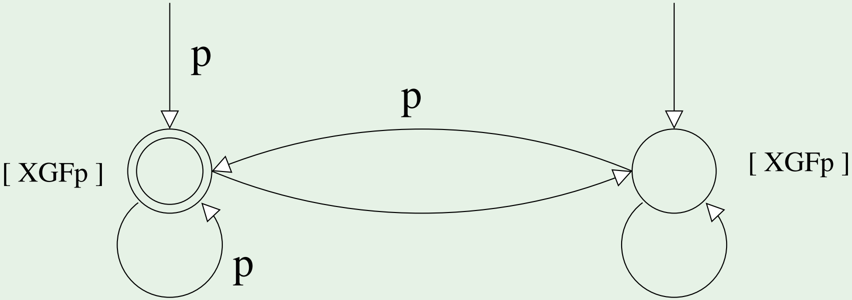
Example: $\varphi = \mathbf{GF}p$

$$
\begin{aligned}
&\mathit{Cover}(\{\mathbf{GF}p\}) \\
&\quad = \mathit{Expand}(\{\mathbf{GF}p\}, \langle \emptyset, \emptyset, \emptyset \rangle) \\
&\quad = \mathit{Expand}(\{\mathbf{F}p\}, \langle \emptyset, \{\mathbf{GF}p\}, \{\mathbf{GF}p\} \rangle) \\
&\quad = \mathit{Expand}(\{\}, \langle \emptyset, \{\mathbf{GF}p, \mathbf{F}p\}, \{\mathbf{GF}p, \mathbf{F}p\} \rangle) \cup \mathit{Expand}(\{p\}, \langle \{\}, \{\mathbf{GF}p\}, \{\mathbf{GF}p, \mathbf{F}p\} \rangle) \\
&\quad = \mathit{Expand}(\{\}, \langle \emptyset, \{\mathbf{GF}p, \mathbf{F}p\}, \{\mathbf{GF}p, \mathbf{F}p\} \rangle) \cup \mathit{Expand}(\{\}, \langle \{p\}, \{\mathbf{GF}p\}, \{\mathbf{GF}p, \mathbf{F}p, p\} \rangle) \\
&\quad = \{\langle \emptyset, \{\mathbf{GF}p, \mathbf{F}p\}, \{\mathbf{GF}p, \mathbf{F}p\} \rangle\} \cup \{\langle \{p\}, \{\mathbf{GF}p\}, \{\mathbf{GF}p, \mathbf{F}p, p\} \rangle\}
\end{aligned}
$$

Note: $\mathbf{GF}p \wedge \mathbf{F}p \iff \mathbf{GF}p$, s.t. $\mathit{Cover}(\mathbf{GF}p \wedge \mathbf{F}p) = \mathit{Cover}(\mathbf{GF}p)$

Example: **GF**p

- Let $s_1 =_{def} \langle \{p\}, \{\mathbf{GF}p\}, \{\mathbf{GF}p, \mathbf{F}p, p\} \rangle$, $s_2 =_{def} \langle \emptyset, \{\mathbf{GF}p, \mathbf{F}p\}, \{\mathbf{GF}p, \mathbf{F}p\} \rangle$,
- $Q = \{s_1, s_2\}$,
- $Q_0 = \{s_1, s_2\}$,
- $T: \begin{aligned} s_1 &\rightarrow \{s_1, s_2\}, \\ s_2 &\rightarrow \{s_1, s_2\} \end{aligned}$
- $FT = \langle F_1 \rangle$ where $F_1 = \{s_1\}$.



NBAs of disjunctions of formulas

Remark

If $\varphi \overset{\text{def}}{=} (\varphi_1 \vee \varphi_2)$ and $A_{\varphi_1}, A_{\varphi_2}$ are NBAs encoding φ_1 and φ_2 resp., then $\mathcal{L}(\varphi) = \mathcal{L}(\varphi_1) \cup \mathcal{L}(\varphi_2)$, so that $A_\varphi \overset{\text{def}}{=} A_{\varphi_1} \cup A_{\varphi_2}$ is an NBA encoding φ

- A_φ non necessarily the smallest/best NBA encoding φ

Example

Let $\varphi \overset{\text{def}}{=} (\mathbf{GF}p \rightarrow \mathbf{GF}q)$, i.e., $\varphi \equiv (\mathbf{FG}\neg p \vee \mathbf{GF}q)$.
Then $A_{\mathbf{FG}\neg p} \cup A_{\mathbf{GF}q}$ encodes φ:

NBAs of disjunctions of formulas

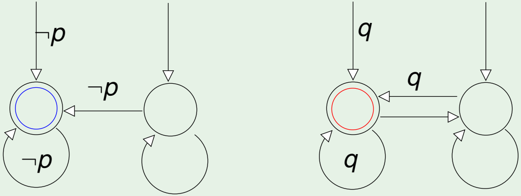
Remark

If $\varphi \stackrel{\text{def}}{=} (\varphi_1 \vee \varphi_2)$ and $A_{\varphi_1}, A_{\varphi_2}$ are NBAs encoding φ_1 and φ_2 resp., then $\mathcal{L}(\varphi) = \mathcal{L}(\varphi_1) \cup \mathcal{L}(\varphi_2)$, so that $A_\varphi \stackrel{\text{def}}{=} A_{\varphi_1} \cup A_{\varphi_2}$ is an NBA encoding φ

- A_φ non necessarily the smallest/best NBA encoding φ

Example

Let $\varphi \stackrel{\text{def}}{=} (\mathbf{GF}p \rightarrow \mathbf{GF}q)$, i.e., $\varphi \equiv (\mathbf{FG}\neg p \vee \mathbf{GF}q)$.
Then $A_{\mathbf{FG}\neg p} \cup A_{\mathbf{GF}q}$ encodes φ:



Suggested Exercises:

- Find an NBA encoding:
  - p
  - $(p \wedge q) \vee (\neg p \wedge \neg q)$
  - $\mathbf{F}p$
  - $\mathbf{G}p$
  - $p\mathbf{R}q$
  - $(\mathbf{GF}p \wedge \mathbf{GF}q) \to \mathbf{G}r$

Outline

1 Büchi Automata

2 The Automata-Theoretic Approach to LTL Reasoning
  - General Ideas
  - Language-Emptiness Checking of Büchi Automata
  - From Kripke Models to Büchi Automata
  - From LTL Formulas to Büchi Automata
  - Complexity

3 Exercises

Automata-Theoretic LTL Model Checking: Complexity

Four steps:

(i) Compute A_M:

$|A_M| = O(|M|)$

(ii) Compute A_φ:

$|A_\varphi| = O(2^{|\varphi|})$

(iii) Compute the product $A_M \times A_\varphi$:

$|A_M \times A_\varphi| = |A_M| \cdot |A_\varphi| = O(|M| \cdot 2^{|\varphi|})$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$:

$O(|A_M \times A_\varphi|) = O(|M| \cdot 2^{|\varphi|})$

$\Longrightarrow$ The complexity of LTL M.C. grows linearly wrt. the size of the model M and exponentially wrt. the size of the property φ

Automata-Theoretic LTL Model Checking: Complexity

Four steps:

(i) Compute A_M:
$|A_M| = O(|M|)$

(ii) Compute A_φ:
$|A_\varphi| = O(2^{|\varphi|})$

(iii) Compute the product $A_M \times A_\varphi$:
$|A_M \times A_\varphi| = |A_M| \cdot |A_\varphi| = O(|M| \cdot 2^{|\varphi|})$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$:
$O(|A_M \times A_\varphi|) = O(|M| \cdot 2^{|\varphi|})$

$\Longrightarrow$ The complexity of LTL M.C. grows linearly wrt. the size of the model M and exponentially wrt. the size of the property φ

Automata-Theoretic LTL Model Checking: Complexity

Four steps:

(i) Compute A_M:
$|A_M| = O(|M|)$

(ii) Compute A_φ:
$|A_\varphi| = O(2^{|\varphi|})$

(iii) Compute the product $A_M \times A_\varphi$:
$|A_M \times A_\varphi| = |A_M| \cdot |A_\varphi| = O(|M| \cdot 2^{|\varphi|})$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$:
$O(|A_M \times A_\varphi|) = O(|M| \cdot 2^{|\varphi|})$

$\Longrightarrow$ The complexity of LTL M.C. grows linearly wrt. the size of the model M and exponentially wrt. the size of the property φ

Automata-Theoretic LTL Model Checking: Complexity

Four steps:

(i) Compute A_M:
$|A_M| = O(|M|)$

(ii) Compute A_φ:
$|A_\varphi| = O(2^{|\varphi|})$

(iii) Compute the product $A_M \times A_\varphi$:
$|A_M \times A_\varphi| = |A_M| \cdot |A_\varphi| = O(|M| \cdot 2^{|\varphi|})$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$:
$O(|A_M \times A_\varphi|) = O(|M| \cdot 2^{|\varphi|})$

$\Longrightarrow$ The complexity of LTL M.C. grows linearly wrt. the size of the model M and exponentially wrt. the size of the property φ

Automata-Theoretic LTL Model Checking: Complexity

Four steps:

(i) Compute A_M:
$|A_M| = O(|M|)$

(ii) Compute A_φ:
$|A_\varphi| = O(2^{|\varphi|})$

(iii) Compute the product $A_M \times A_\varphi$:
$|A_M \times A_\varphi| = |A_M| \cdot |A_\varphi| = O(|M| \cdot 2^{|\varphi|})$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$:
$O(|A_M \times A_\varphi|) = O(|M| \cdot 2^{|\varphi|})$

$\Longrightarrow$ The complexity of LTL M.C. grows linearly wrt. the size of the model M and exponentially wrt. the size of the property φ

Automata-Theoretic LTL Model Checking: Complexity

Four steps:

(i) Compute A_M:
$|A_M| = O(|M|)$

(ii) Compute A_φ:
$|A_\varphi| = O(2^{|\varphi|})$

(iii) Compute the product $A_M \times A_\varphi$:
$|A_M \times A_\varphi| = |A_M| \cdot |A_\varphi| = O(|M| \cdot 2^{|\varphi|})$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$:
$O(|A_M \times A_\varphi|) = O(|M| \cdot 2^{|\varphi|})$

$\Longrightarrow$ The complexity of LTL M.C. grows linearly wrt. the size of the model M and exponentially wrt. the size of the property φ

Automata-Theoretic LTL Model Checking: Complexity

Four steps:

(i) Compute A_M:
$|A_M| = O(|M|)$

(ii) Compute A_φ:
$|A_\varphi| = O(2^{|\varphi|})$

(iii) Compute the product $A_M \times A_\varphi$:
$|A_M \times A_\varphi| = |A_M| \cdot |A_\varphi| = O(|M| \cdot 2^{|\varphi|})$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$:
$O(|A_M \times A_\varphi|) = O(|M| \cdot 2^{|\varphi|})$

$\Longrightarrow$ The complexity of LTL M.C. grows linearly wrt. the size of the model *M* and exponentially wrt. the size of the property φ

Automata-Theoretic LTL Model Checking: Complexity

Four steps:

(i) Compute A_M:
$|A_M| = O(|M|)$

(ii) Compute A_φ:
$|A_\varphi| = O(2^{|\varphi|})$

(iii) Compute the product $A_M \times A_\varphi$:
$|A_M \times A_\varphi| = |A_M| \cdot |A_\varphi| = O(|M| \cdot 2^{|\varphi|})$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$:
$O(|A_M \times A_\varphi|) = O(|M| \cdot 2^{|\varphi|})$

$\Longrightarrow$ The complexity of LTL M.C. grows linearly wrt. the size of the model M and exponentially wrt. the size of the property φ

Automata-Theoretic LTL Model Checking: Complexity

Four steps:

(i) Compute A_M:
$|A_M| = O(|M|)$

(ii) Compute A_φ:
$|A_\varphi| = O(2^{|\varphi|})$

(iii) Compute the product $A_M \times A_\varphi$:
$|A_M \times A_\varphi| = |A_M| \cdot |A_\varphi| = O(|M| \cdot 2^{|\varphi|})$

(iv) Check the emptiness of $\mathcal{L}(A_M \times A_\varphi)$:
$O(|A_M \times A_\varphi|) = O(|M| \cdot 2^{|\varphi|})$

$\Longrightarrow$ The complexity of LTL M.C. grows linearly wrt. the size of the model M and exponentially wrt. the size of the property φ

Final Remarks

- Büchi automata are in general more expressive than LTL!

⟹ some tools (e.g., Spin) allow specifications to be expressed directly as NBAs

⟹ complementation of NBA relevanant in general

- For every LTL formula, there are many possible equivalent NBAs

⟹ lots of research for finding “the best” conversion algorithm

- Performing the product and checking emptiness very relevant

⟹ lots of techniques developed (e.g., partial order reduction)

⟹ lots on ongoing research

Final Remarks

- Büchi automata are in general more expressive than LTL!

⟹ some tools (e.g., Spin) allow specifications to be expressed directly as NBAs

⟹ complementation of NBA relevanant in general

- For every LTL formula, there are many possible equivalent NBAs

⟹ lots of research for finding “the best” conversion algorithm

- Performing the product and checking emptiness very relevant

⟹ lots of techniques developed (e.g., partial order reduction)

⟹ lots on ongoing research

Final Remarks

- Büchi automata are in general more expressive than LTL!

$\Longrightarrow$ some tools (e.g., Spin) allow specifications to be expressed directly as NBAs

$\Longrightarrow$ complementation of NBA relevanant in general

- For every LTL formula, there are many possible equivalent NBAs

$\Longrightarrow$ lots of research for finding “the best” conversion algorithm

- Performing the product and checking emptiness very relevant

$\Longrightarrow$ lots of techniques developed (e.g., partial order reduction)

$\Longrightarrow$ lots on ongoing research

Outline

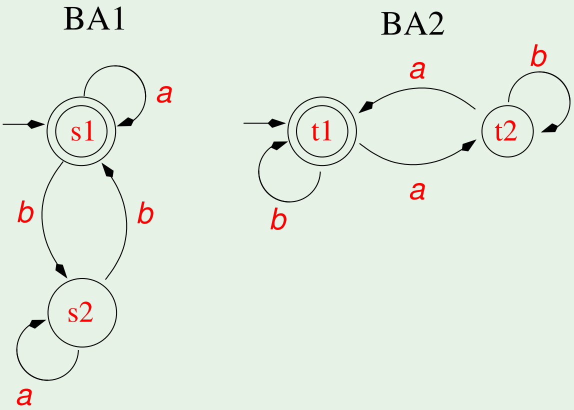
Ex: Product of Büchi automata

Given the following two Büchi automata (doubly-circled states represent accepting states, *a*, *b* are labels):

Write the product Büchi automaton $BA1 \times BA2$.

Ex: Product of Büchi automata

Given the following two Büchi automata (doubly-circled states represent accepting states, *a*, *b* are labels):



Write the product Büchi automaton *BA*1 × *BA*2.

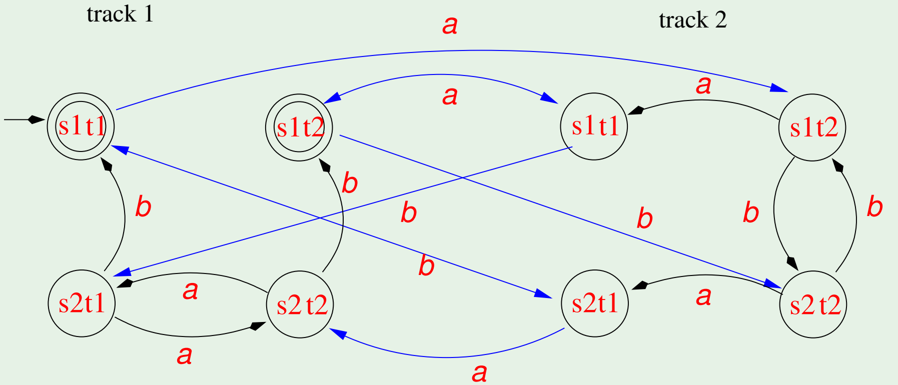
Ex: Product of Büchi automata

[Solution: The product is:

]

Ex: Product of Büchi automata

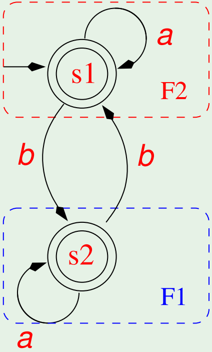
[Solution: The product is:



]

Ex: De-generalization of Büchi Automata

Given the following generalized Büchi automaton $A \overset{\text{def}}{=} \langle Q, \Sigma, \delta, I, FT \rangle$, with two sets of accepting states $FT \overset{\text{def}}{=} \{F1, F2\}$ s.t. $F1 \overset{\text{def}}{=} \{s2\}, F2 \overset{\text{def}}{=} \{s1\}$:



convert it into an equivalent plain Büchi automaton.

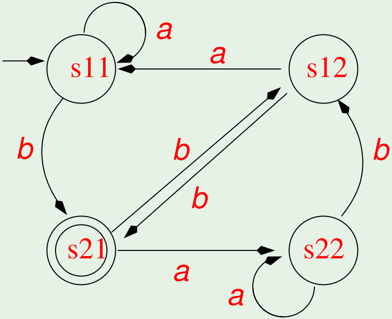
Ex: De-generalization of Büchi Automata

[Solution: The result is:

]

Ex: De-generalization of Büchi Automata

[Solution: The result is:



]

Ex: Construction of Büchi Automata

Consider the LTL formula $\varphi \overset{\text{def}}{=} (\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q)$.

Ex: Construction of Büchi Automata

Consider the LTL formula $\varphi \overset{\text{def}}{=} (\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q)$.

(*a*) rewrite φ into Negative Normal Form

Ex: Construction of Büchi Automata

Consider the LTL formula $\varphi \overset{\text{def}}{=} (\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q)$.

(a) rewrite φ into Negative Normal Form

[Solution: $(\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q) \Longrightarrow (\neg\mathbf{G}\neg p) \vee (p\mathbf{U}q) \Longrightarrow (\mathbf{F}p) \vee (p\mathbf{U}q)$]

Ex: Construction of Büchi Automata

Consider the LTL formula $\varphi \stackrel{\text{def}}{=} (\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q)$.

(a) rewrite φ into Negative Normal Form

[Solution: $(\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q) \Longrightarrow (\neg\mathbf{G}\neg p) \vee (p\mathbf{U}q) \Longrightarrow (\mathbf{F}p) \vee (p\mathbf{U}q)$]

(b) find the initial states of a corresponding Buchi automaton (for each state, define the labels of the incoming arcs and the “next” section.)

Ex: Construction of Büchi Automata

Consider the LTL formula $\varphi \overset{\text{def}}{=} (\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q)$.

(*a*) rewrite φ into Negative Normal Form

[Solution: $(\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q) \Longrightarrow (\neg\mathbf{G}\neg p) \vee (p\mathbf{U}q) \Longrightarrow (\mathbf{F}p) \vee (p\mathbf{U}q)$]

(*b*) find the initial states of a corresponding Buchi automaton (for each state, define the labels of the incoming arcs and the “next” section.)

[Solution: Applying tableaux rules we obtain: $p \vee \mathbf{XF}p \vee q \vee (p \wedge \mathbf{X}(p\mathbf{U}q))$, which is already in disjunctive normal form. This correspond to the following four initial states:

]

Ex: Construction of Büchi Automata

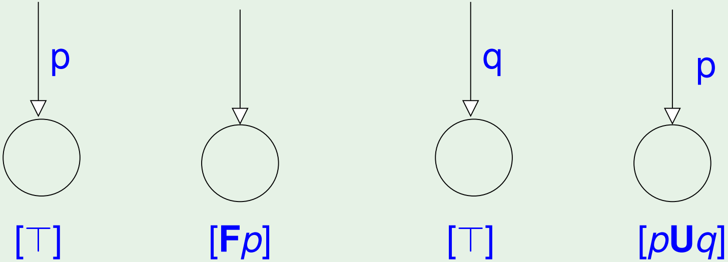
Consider the LTL formula $\varphi \overset{\text{def}}{=} (\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q)$.

(a) rewrite φ into Negative Normal Form

[Solution: $(\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q) \Longrightarrow (\neg\mathbf{G}\neg p) \vee (p\mathbf{U}q) \Longrightarrow (\mathbf{F}p) \vee (p\mathbf{U}q)$]

(b) find the initial states of a corresponding Buchi automaton (for each state, define the labels of the incoming arcs and the “next” section.)

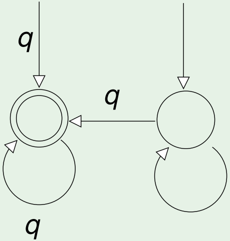
[Solution: Applying tableaux rules we obtain: $p \vee \mathbf{XF}p \vee q \vee (p \wedge \mathbf{X}(p\mathbf{U}q))$, which is already in disjunctive normal form. This correspond to the following four initial states:



]

Ex: Büchi automaton

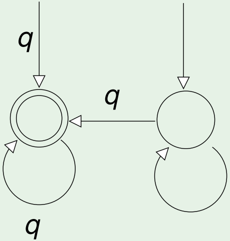
Given the following Büchi automaton BA (doubly-circled states represent accepting states):



Say which of the following sentences are true and which are false.

Ex: Büchi automaton

Given the following Büchi automaton BA (doubly-circled states represent accepting states):

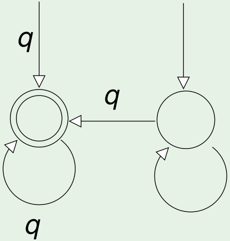


Say which of the following sentences are true and which are false.

(*a*) BA accepts all and only the paths verifying **GF**q.

Ex: Büchi automaton

Given the following Büchi automaton BA (doubly-circled states represent accepting states):

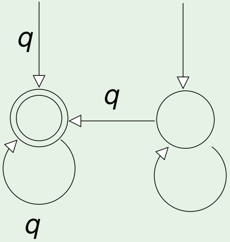


Say which of the following sentences are true and which are false.

(*a*) BA accepts all and only the paths verifying **GF**q. [Solution: false]

Ex: Büchi automaton

Given the following Büchi automaton BA (doubly-circled states represent accepting states):



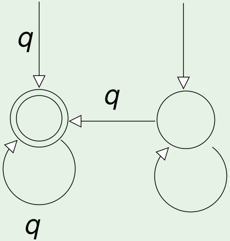
Say which of the following sentences are true and which are false.

(*a*) BA accepts all and only the paths verifying **GF**q. [Solution: false]

(*b*) BA accepts all and only the paths verifying **FG**q.

Ex: Büchi automaton

Given the following Büchi automaton BA (doubly-circled states represent accepting states):



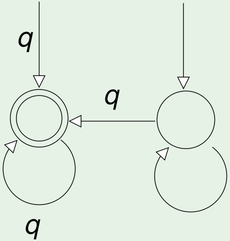
Say which of the following sentences are true and which are false.

(*a*) BA accepts all and only the paths verifying **GF**q. [Solution: false]

(*b*) BA accepts all and only the paths verifying **FG**q. [Solution: true]

Ex: Büchi automaton

Given the following Büchi automaton BA (doubly-circled states represent accepting states):



Say which of the following sentences are true and which are false.

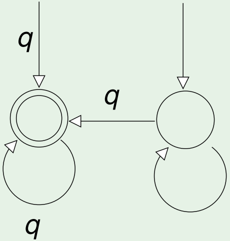
(*a*) BA accepts all and only the paths verifying **GF**q. [Solution: false]

(*b*) BA accepts all and only the paths verifying **FG**q. [Solution: true]

(*c*) BA accepts only paths verifying **F**q, but not all of them.

Ex: Büchi automaton

Given the following Büchi automaton BA (doubly-circled states represent accepting states):



Say which of the following sentences are true and which are false.

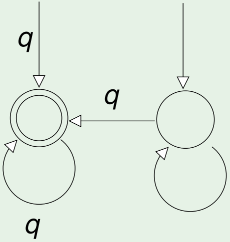
(a) BA accepts all and only the paths verifying **GF**q. [Solution: false]

(b) BA accepts all and only the paths verifying **FG**q. [Solution: true]

(c) BA accepts only paths verifying **F**q, but not all of them. [Solution: true]

Ex: Büchi automaton

Given the following Büchi automaton BA (doubly-circled states represent accepting states):



Say which of the following sentences are true and which are false.

(a) BA accepts all and only the paths verifying **GF**q. [Solution: false]

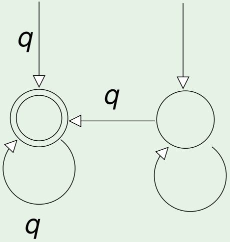
(b) BA accepts all and only the paths verifying **FG**q. [Solution: true]

(c) BA accepts only paths verifying **F**q, but not all of them. [Solution: true]

(d) BA accepts all the paths verifying **F**q, but not only them.

Ex: Büchi automaton

Given the following Büchi automaton BA (doubly-circled states represent accepting states):



Say which of the following sentences are true and which are false.

(*a*) BA accepts all and only the paths verifying **GF**q. [Solution: false]

(*b*) BA accepts all and only the paths verifying **FG**q. [Solution: true]

(*c*) BA accepts only paths verifying **F**q, but not all of them. [Solution: true]

(*d*) BA accepts all the paths verifying **F**q, but not only them. [Solution: false]