Introduction to Formal Methods
Chapter 08: Automata-theoretic LTL Model Checking

Roberto Sebastiani and Stefano Tonetta

DISI, Università di Trento, Italy – roberto.sebastiani@unitn.it
URL: http://disi.unitn.it/rseba/DIDATTICA/fm2020/
Teaching assistant: Enrico Magnago – enrico.magnago@unitn.it

CDLM in Informatica, academic year 2019-2020

last update: Monday 18th May, 2020, 14:49

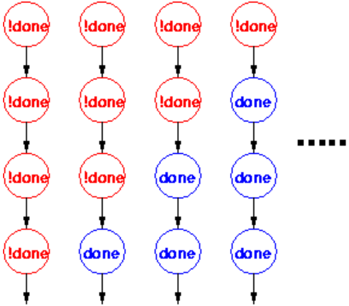
Copyright notice: *some material (text, figures) displayed in these slides is courtesy of R. Alur, M. Benerecetti, A. Cimatti, M. Di Natale, P. Pandya, M. Pistore, M. Roveri, and S.Tonetta, who detain its copyright. Some exampes displayed in these slides are taken from [Clarke, Grunberg & Peled, "Model 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*

Outline

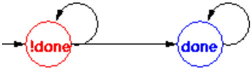
System's computations

- The behaviors (computations) of a system can be seen as sequences of assignments to propositions.

```
MODULE main
VAR  done: Boolean;
ASSIGN
   init(done):=0;
   next(done):= case
      !done: {0,1};
      done: done;
    esac;
```



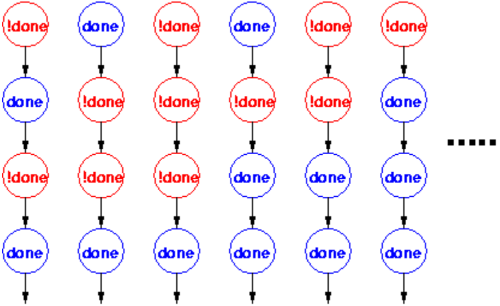
- Since the state space is finite, the set of computations can be represented by a finite automaton.

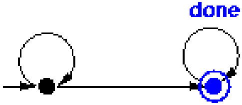


or

Correct computations

- Some computations are correct and others are not acceptable.
- We can build an automaton for the set of all acceptable computations.
- Example: eventually, done will be true forever (**FG**done).





Language Containment Problem

- Solution to the verification problem
  $\Longrightarrow$ Check if language of the system automaton is contained in the language accepted by the property automaton.
- The language containment problem is the problem of deciding if a language is a subset of another language.

$$\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2) \Longleftrightarrow \mathcal{L}(A_1) \cap \overline{\mathcal{L}(A_2)} = \{\}$$

- In order to solve the language containment problem, we need to know:
  (i) how to complement an automaton,
  (ii) how to intersect two automata,
  (iii) how to check the language emptiness of an automaton.

Finite Word Languages

- An Alphabet Σ is a collection of symbols (letters).
  E.g. $\Sigma = \{a, b\}$.
- A finite word is a finite sequence of letters. (E.g. *aabb*.)
  The set of all finite words is denoted by Σ^*.
- A language U is a set of words, i.e. $U \subseteq \Sigma^*$.
  Example: Words over $\Sigma = \{a, b\}$ with equal number of a's and b's.
  (E.g. *aabb* or *abba*.)
- Language recognition problem: determine whether a word belongs to a language.
- Automata are computational devices able to solve language recognition problems.

Finite-State Automata

- Basic model of computational systems with finite memory.
- Widely applicable
  - Embedded System Controllers.
    Languages: Ester-el, Lustre, Verilog.
  - Synchronous Circuits.
  - Regular Expression Pattern Matching
    Grep, Lex, Emacs.
  - Protocols
    Network Protocols
    Architecture: Bus, Cache Coherence, Telephony,...

Notation

$a, b \in \Sigma$ finite alphabet.
$u, v, w \in \Sigma^*$ finite words.

- ϵ empty word.
- $u.v$ concatenation.
- $u^i = u.u. .u$ repeated i-times.

$U, V \subseteq \Sigma^*$ Finite word languages.

Finite-State Automata Definition

Definition

A Nondeterministic Finite-State Automaton (NFA) 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 final states.

$\delta \subseteq Q \times \Sigma \times Q$ transition relation (edges).

We use $q \xrightarrow{a} q'$ to denote $(q, a, q') \in \delta$.

Definition

A Deterministic Finite-State Automaton (DFA) is a NFA s.t.:

$\delta : Q \times \Sigma \rightarrow Q$ is a total function

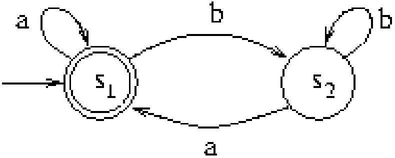
Single initial state $I = \{q_0\}$.

Regular Languages

- A run of NFA A on $u = a_0, a_1, \ldots, a_{n-1}$ is a finite sequence of states $q_0, q_1, \ldots, q_n$ s.t. $q_0 \in I$ and $q_i \xrightarrow{a_i} q_{i+1}$ for $0 \leq i < n$.
- An accepting run is one where $q_n \in F$.
- The language accepted by A is
  $\mathcal{L}(A) = \{u \in \Sigma^* \mid A \text{ has an accepting run on } u\}$
- The languages accepted by a NFA are called regular languages.

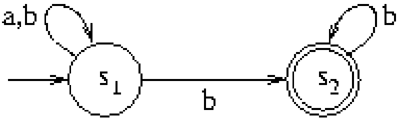
Finite-State Automata: examples

- The DFA A_1 over $\Sigma = \{a, b\}$:



Recognizes words which do not end in *b*.

- The NFA A_2 over $\Sigma = \{a, b\}$:



Recognizes words which end in *b*.

Determinisation

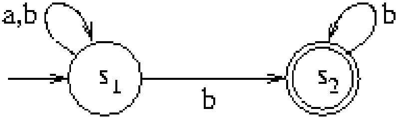
Theorem (determinisation)

Given a NFA A we can construct a DFA A' s.t. $\mathcal{L}(A) = \mathcal{L}(A')$.
Size: $|A'| = 2^{O(|A|)}$.

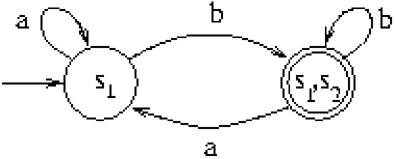
- Each state of A' corresponds to a set $\{s_1, ..., s_j\}$ of states in A ($Q' \subseteq 2^Q$), with the intended meaning that :
  - A' is in the state $\{s_1, .., s_j\}$ if A is in one of the states $s_1, ..., s_j$
- The (unique) initial state is $I' =_{def} \{s_i \mid s_i \in I\}$
- The deterministic transition relation $\delta' : 2^Q \times \Sigma \longmapsto 2^Q$ is
  - $\{s\} \xrightarrow{a} \{s_i \mid s \xrightarrow{a} s_i\}$
  - $\{s_1, ..., s_j, ..., s_n\} \xrightarrow{a} \bigcup_{j=1}^{n} \{s_i \mid s_j \xrightarrow{a} s_i\}$
- The set of final states F' is such that $\{s_1, ..., s_n\} \in F'$ iff $s_i \in F$ for some $i \in \{1, ..., n\}$

Determinisation [cont.]

- NFA A_2: Words which end in b.



- A_2 can be determinised into the automaton DA_2 below. (#States = 2^Q.)



Closure Properties

Theorem (Boolean closure)

Given NFA A_1, A_2 over Σ we can construct NFA A over Σ s.t.

- $\mathcal{L}(A) = \overline{\mathcal{L}(A_1)}$ (Complement). $|A| = 2^{O(|A_1|)}$.
- $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$ (union). $|A| = |A_1| + |A_2|$.
- $\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ (intersection). $|A| \leq |A_1| \cdot |A_2|$.

Complementation of a NFA

A NFA $A = (Q, \Sigma, \delta, I, F)$ is complemented by:

- determinising it into a DFA $A' = (Q', \Sigma', \delta', I', F')$
- complementing it: $\overline{A'} = (Q', \Sigma', \delta', I', \overline{F'})$
- $|\overline{A'}| = |A'| = 2^{O(|A|)}$

Union of two NFAs

Definition: union of NFAs

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

Synchronous Product Construction

Definition: product of NFAs

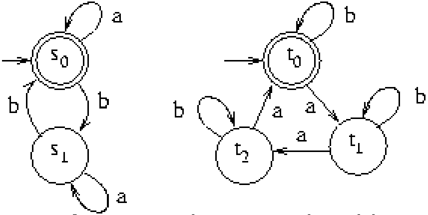
Let $A_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$.
Then, $A_1 \times A_2 = (Q, \Sigma, \delta, I, F)$ where

- $Q = Q_1 \times Q_2$,
- $I = I_1 \times I_2$,
- $F = F_1 \times F_2$,
- $\langle p, q \rangle \xrightarrow{a} \langle p', q' \rangle$ iff $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$.

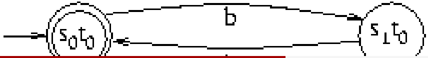
Theorem

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

Example



- A_1 recognizes words with an even number of b's.
- A_2 recognizes words with a number of a's multiple of 3.
- The Product Automaton $A_1 \times A_2$ with $F = \{s_0, t_0\}$.



Regular Expressions

- Syntax: $\emptyset \mid \epsilon \mid a \mid reg_1.reg_2 \mid reg_1|reg_2 \mid reg^*$.
- Every regular expression *reg* denotes a language $\mathcal{L}(reg)$.
- Example: $a^*.(b|bb).a^*$. The words with either 1 *b* or 2 consecutive *b*'s.

Theorem

For every regular expression *reg* we can construct a language equivalent NFA of size $O(|reg|)$.

Theorem

For every DFA A we can construct a language equivalent regular expression $reg(A)$.

Infinite Word Languages

Modeling infinite computations of reactive systems.

- 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:

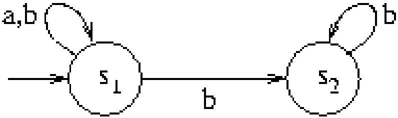
omega words $\alpha, \beta, \gamma \in \Sigma^\omega$.

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 $\rho \in Q^\omega$. 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

$A = (Q, \Sigma, \delta, I, F)$, where $F \subseteq Q$ is the set of accepting states.

- 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

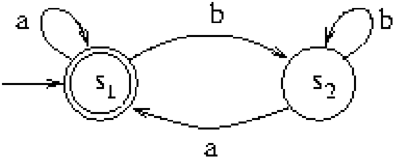
  $$Inf(\rho) \cap F \neq \emptyset.$$

- The language accepted by A

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

Büchi Automaton: Example

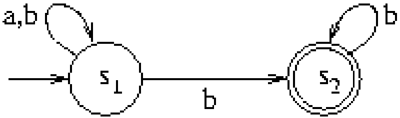
Let $\Sigma = \{a, b\}$.
Let a Deterministic Büchi Automaton (DBA) A_1 be



- 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



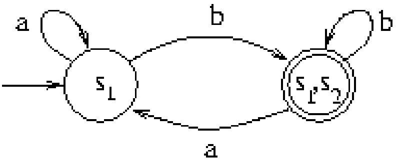
With $F = \{s_2\}$, the automaton A_2 recognizes words with finitely many a. Thus, $\mathcal{L}(A_2) = \overline{\mathcal{L}(A_1)}$.

Deterministic vs. Nondeterministic Büchi Automata

Theorem

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

The subset construction does not work:
let DA_2 be



- DA_2 is not equivalent to A_2
  (e.g., it recognizes $(b.a)^\omega$)

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)

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

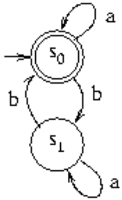
- $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.
- $|A_1 \times A_2| \leq 2 \cdot |A_1| \cdot |A_2|$.

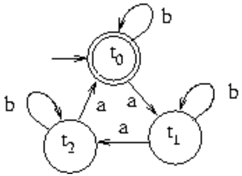
Product of NBAs: Intuition

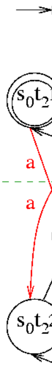
- The automaton remembers two tracks, one for each source NBA, and it points to one of the two tracks
- As soon as it goes through an accepting state of the current track, it switches to the other track
  $\Longrightarrow$ in order to visit infinitely often a state in F (i.e., F_1), it must visit infinitely often some state also in F_2
- Important subcase: If $F_2 = Q_2$, then

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

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.

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 $(i + 1) mod\ K$ mode.

De-generalization of a generalized NBA

Definition: De-generalization of a generalized NBA

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

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

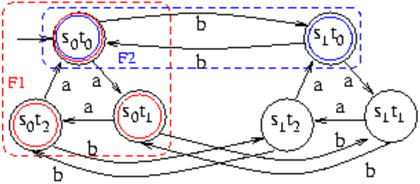
δ' is s.t., for every $i \in [1, ..., K]$:

$$\langle p, i\rangle \xrightarrow{a} \langle q, i\rangle \quad \text{iff} \quad p \xrightarrow{a} q \in \delta \quad \text{and} \quad p \notin F_i.$$
$$\langle p, i\rangle \xrightarrow{a} \langle q, (i+1) mod\ K\rangle \quad \text{iff} \quad p \xrightarrow{a} q \in \delta \quad \text{and} \quad p \in F_i.$$

Theorem

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

Degeneralizing a Büchi automaton: Example



Omega-regular Expressions

Definition

A 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.

NFA emptiness checking

- Equivalent of finding a final state reachable from an initial state.
- It can be solved with a DFS or a BFS.
- A DFS finds a counterexample on the fly (it is stored in the stack of the procedure).
- A BFS finds a final state reachable with a shortest counterexample, but it requires a further backward search to reproduce the path.
- Complexity: $O(n)$.
- Hereafter, assume w.l.o.g. that there is only one initial state.

NFA Emptiness Checking (cont.)

// returns True if empty language, false otherwise

```
Bool DFS(NFA A) {
   stack S=I;
   Hashtable T=I;
   while S!=∅ {
      v=top(S);
      if v∈F return False
      if ∃w s.t. w∈δ(v) && T(w)==0 {
         hash(w,T);
         push(w,S);
      } else
         pop(S);
   }
   return True;
}
```

NBA emptiness checking

- Equivalent of finding an accepting cycle reachable from an initial state.
- A naive algorithm:
  (i) a DFS finds the final 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) 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.

Double Nested DFS algorithm

- Double Nested DFS [Courcoubetis, Vardi, Wolper, Yannakakis, CAV'90]
  - two Hash tables:
    - `T1`: reachable states
    - `T2`: states reachable from a reachable final state
  - two stacks:
    - `S1`: current branch of states reachable
    - `S2`: current branch of states reachable from final state f
  - two nested DFS's:
    - `DFS1` looks for a path from an initial state to a cycle starting from an accepting state
    - `DFS2` looks for a cycle starting from an accepting state
  - It stops as soon as it finds a counterexample.
  - The counterexample is given by the stack of `DFS2` (an accepting cycle) preceded by the stack of `DFS1` (a path from an initial state to the cycle).

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))
            return False;
   }    }
   return True;
}
```

Double Nested DFS - Second DFS

```
Bool DFS2(state f, stack & S, Hashtable & T, NBA A) {
   hash(f,T);
   S = {f}
   while S!=∅ {
      v=top(S);
      if f∈δ(v) return False;
      if ∃w s.t. w∈δ(v) && T(w)==0 {
         hash(w);
         push(w);
      } else pop(S);
   }
   return True;
}
```

Remark: `T` passed by reference, is not reset at each call of `DFS2` !

Double nested DFS: intuition

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

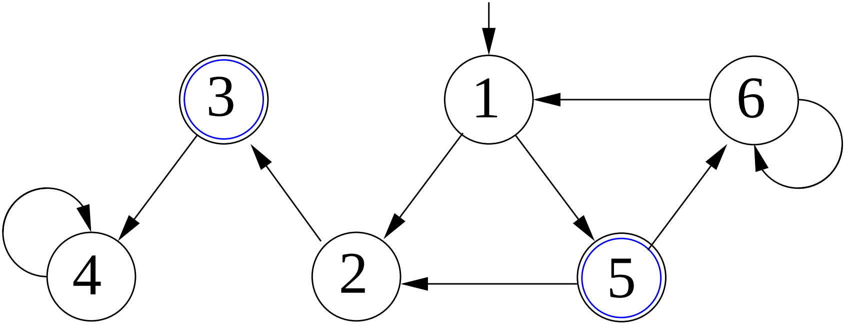
- suppose $DFS2$ is invoked on f_j before than on f_i

$\Longrightarrow$ 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: example



T1

S1

T2

S2

Automata-Theoretic LTL Model Checking

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

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

- 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 M.C. (dual version)

- Let M be a Kripke model and $\varphi \stackrel{\text{def}}{=} \neg\psi$ be an LTL formula

$$
\begin{aligned}
& M \models \mathbf{E}\varphi \\
\Longleftrightarrow\ & M \not\models \mathbf{A}\neg\varphi \\
\Longleftrightarrow\ & ... \\
\Longleftrightarrow\ & \mathcal{L}(A_M \times A_\varphi) \neq \emptyset
\end{aligned}
$$

- A_M is a Büchi Automaton equivalent to M (which represents all and only the executions of M)
- A_φ is a Büchi Automaton which represents all and only the paths that satisfy φ

$\Longrightarrow$ $A_M \times A_\varphi$ represents all and only the paths appearing in both A_M and A_φ.

Automata-Theoretic LTL Model Checking

Four steps:

(i) Compute A_M

(ii) Compute A_φ

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

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

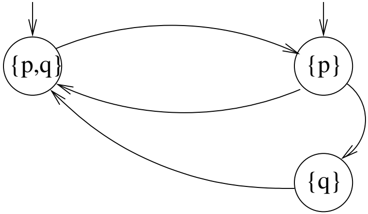
Computing an NBA A_M from a Kripke Structure M

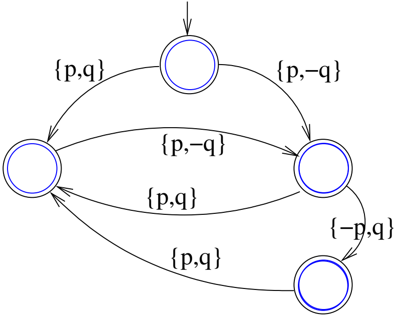
- Transform a Kripke structure $M = \langle S, S_0, R, L, AP\rangle$ into an NBA $A_M = \langle Q, \Sigma, \delta, I, F\rangle$ s.t.:
  - States: $Q := S \cup \{init\}$, *init* being a new initial state
  - Alphabet: $\Sigma := 2^{AP}$
  - Initial State: $I := \{init\}$
  - Accepting States: $F := Q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

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

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

Computing a NBA A_M from a Kripke Structure M: Example

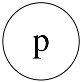




$\Longrightarrow$ Substantially, add one initial state, move labels from states to incoming edges, set all states as accepting states

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.

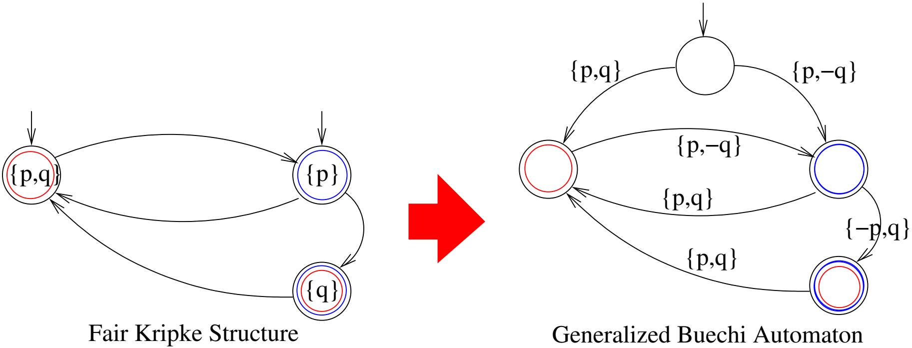
Computing a NBA A_M from a Fair Kripke Structure M

- Transforming a fair K.S. $M = \langle S, S_0, R, L, AP, FT \rangle$, $FT = \{F_1, ..., F_n\}$, into a generalized NBA $A_M = \langle Q, \Sigma, \delta, I, FT' \rangle$ s.t.:
  - States: $Q := S \cup \{init\}$, *init* being a new initial state
  - Alphabet: $\Sigma := 2^{AP}$
  - Initial State: $I := \{init\}$
  - Accepting States: $FT' := FT$
  - Transitions:

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

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

Computing a (Generalized) BA A_M from a Fair Kripke Structure M: Example



$\Longrightarrow$ Substantially, add one initial state, move labels from states to incoming edges, set fair states as accepting states

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 model checking (in other words, every model checking algorithm that verifies the correctness of an LTL formula translates it in some sort of finite-state machine).
- We will translate an LTL formula into a Generalized Büchi Automata (GBA).

Exponential Translation

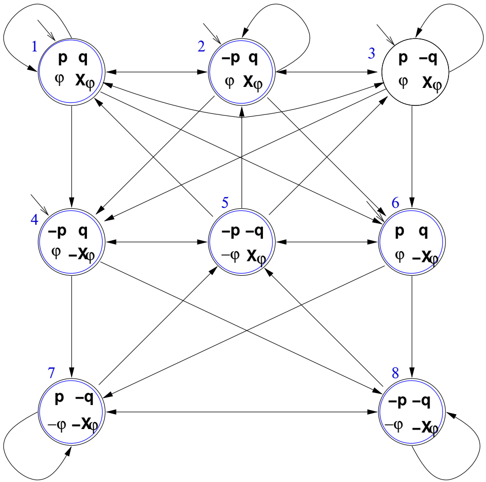
- From φ, create a fair Kripke model, like in chapter 7.
- Convert it into a (Generalized) Büchi Automaton

Remark

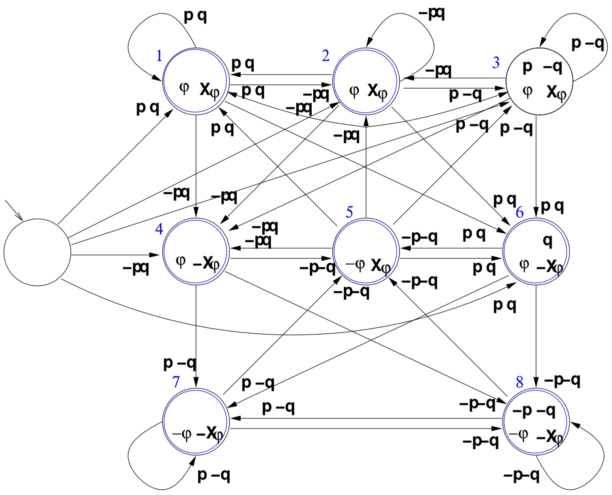
Inefficient: up to $2^{EL(\varphi)}$ states.

- Kripke models require total truth assignments to state variables

Example



Example



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:

$$
\begin{array}{lcl}
\neg(\varphi_1 \vee \varphi_2) & \Longrightarrow & (\neg\varphi_1 \wedge \neg\varphi_2) \\
\neg(\varphi_1 \wedge \varphi_2) & \Longrightarrow & (\neg\varphi_1 \vee \neg\varphi_2) \\
\neg\mathbf{X}\varphi_1 & \Longrightarrow & \mathbf{X}\neg\varphi_1 \\
\neg(\varphi_1\mathbf{U}\varphi_2) & \Longrightarrow & (\neg\varphi_1\mathbf{R}\neg\varphi_2) \\
\neg(\phi_1\mathbf{R}\phi_2) & \Longrightarrow & (\neg\phi_1\mathbf{U}\neg\phi_2)
\end{array}
$$

  $\Longrightarrow$ the resulting formula is expressed in terms of $\vee$, $\wedge$, X, **U**, **R** and literals (Negative Normal Form, NNF).
  - encoding linear if a DAG representation is used
- In the construction of A_φ we now assume that φ is in NNF.

On-the-fly Construction of A_φ (Intuition)

Apply recursively the following steps:

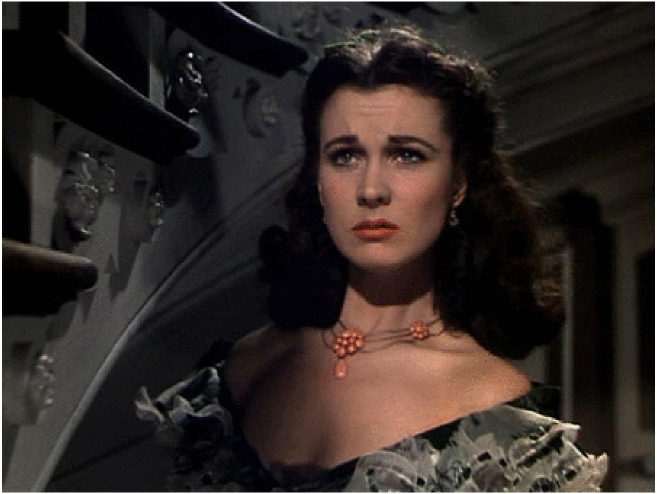
Step 1: Apply the tableau expansion rules to φ

$\psi_1 \mathbf{U} \psi_2 \Longrightarrow \psi_2 \vee (\psi_1 \wedge \mathbf{X}(\psi_1 \mathbf{U} \psi_2))$

$\psi_1 \mathbf{R} \psi_2 \Longrightarrow \psi_2 \wedge (\psi_1 \vee \mathbf{X}(\psi_1 \mathbf{R} \psi_2))$

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, and then push the conjunctions inside the next:

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

- 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 (obligations 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.]

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., $FT \stackrel{\text{def}}{=} \{Q\}$).

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 $\bigwedge_j \psi_j$.
  - Combines steps 1. and 2. of previous slides
  - Expand() defined recursively as follows

On-the-fly Construction of A_ϕ - Expand

Given a set of formulas Φ to expand and a state s, we define the set of states $Expand(\Phi, s)$ recursively as follows:

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

On-the-fly Construction of A_ϕ - Expand

- if $\psi_1 \vee \psi_2 \in \Phi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $$Expand(\Phi, s) = Expand(\Phi \cup \{\psi_1\} \backslash \{\psi_1 \vee \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \vee \psi_2\} \rangle) \cup Expand(\Phi \cup \{\psi_2\} \backslash \{\psi_1 \vee \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \vee \psi_2\} \rangle)$$
  (split s in two copies, process ψ_2 on the first, ψ_1 on the second, add $\psi_1 \vee \psi_2$ to σ)
- if $\psi_1 \mathbf{U} \psi_2 \in \Phi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $$Expand(\Phi, s) = Expand(\Phi \cup \{\psi_1\} \backslash \{\psi_1 \mathbf{U} \psi_2\}, \langle \lambda, \chi \cup \{\psi_1 \mathbf{U} \psi_2\}, \sigma \cup \{\psi_1 \mathbf{U} \psi_2\} \rangle) \cup Expand(\Phi \cup \{\psi_2\} \backslash \{\psi_1 \mathbf{U} \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \mathbf{U} \psi_2\} \rangle)$$
  (split s in two copies and process ψ_1 on the first, ψ_2 on the second, add $\psi_1 \mathbf{U} \psi_2$ to σ)
- if $\psi_1 \mathbf{R} \psi_2 \in \Phi$ and $s = \langle \lambda, \chi, \sigma \rangle$,
  $$Expand(\Phi, s) = Expand(\Phi \cup \{\psi_2\} \backslash \{\psi_1 \mathbf{R} \psi_2\}, \langle \lambda, \chi \cup \{\psi_1 \mathbf{R} \psi_2\}, \sigma \cup \{\psi_1 \mathbf{R} \psi_2\} \rangle) \cup Expand(\Phi \cup \{\psi_1, \psi_2\} \backslash \{\psi_1 \mathbf{R} \psi_2\}, \langle \lambda, \chi, \sigma \cup \{\psi_1 \mathbf{R} \psi_2\} \rangle)$$
  (split s in 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: $\mathbf{F}\psi \stackrel{\text{def}}{=} \top\mathbf{U}\psi$ and $\mathbf{G}\psi \stackrel{\text{def}}{=} \bot\mathbf{R}\psi$

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

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 $A_\phi = (Q, Q_0, \Sigma, L, T, FT)$ as follows:

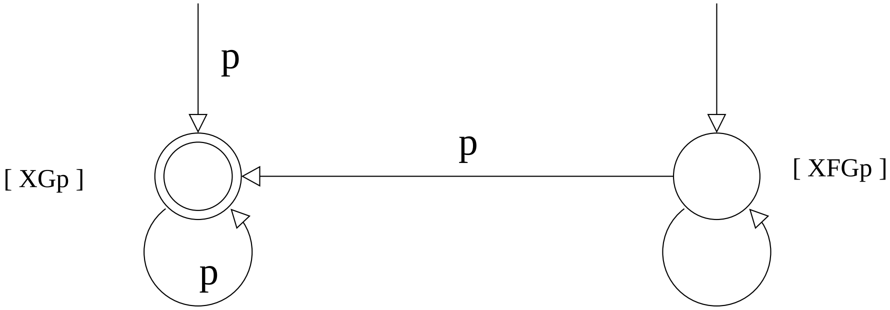
- $\Sigma \;=\; 2^{vars(\phi)}$
- Q is the smallest set such that
  - $Cover(\{\phi\}) \subseteq Q$
  - if $\langle \lambda, \chi, \sigma \rangle \in Q$, then $Cover(\chi) \in Q$
- $Q_0 = Cover(\{\phi\})$.
- $L(\langle \lambda, \chi, \sigma \rangle) = \{a \in \Sigma | a \models \lambda\}$
- $(s, s') \in T$ iff, $s = \langle \lambda, \chi, \sigma \rangle$ and $s' \in Cover(\chi)$
- $FT = \langle F_1, F_2, ..., F_k \rangle$ where, for all $(\psi_i \mathbf{U} \phi_i)$ occurring positively in ϕ, $F_i \;=\; \{\langle \lambda, \chi, \sigma \rangle \in Q \;\mid\; (\psi_i \mathbf{U} \phi_i) \notin \sigma \;\; or \;\; \phi_i \in \sigma\}$.
  (If there is no U-subformulas, then $FT \stackrel{\text{def}}{=} \{Q\}$).

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

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

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

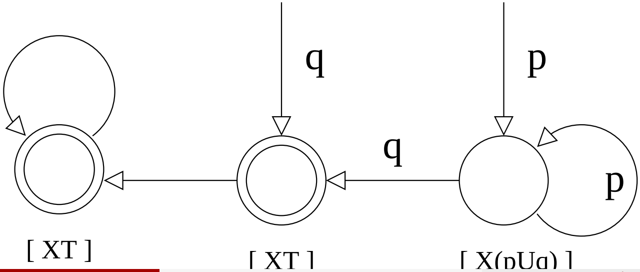


Example: $\phi = p\mathbf{U}q$

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

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

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



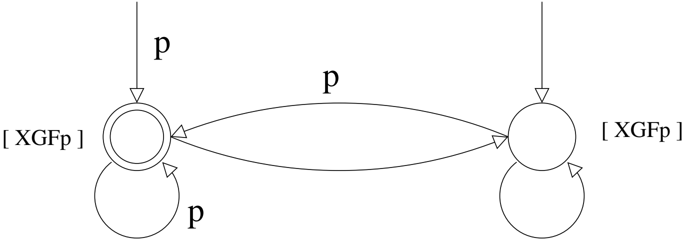
Example: $\phi = \mathbf{GF}p$

$$
\begin{array}{rl}
& \mathit{Cover}(\{\mathbf{GF}p\}) \\
= & E(\{\mathbf{GF}p\}, \langle \emptyset, \emptyset, \emptyset \rangle) \\
= & E(\{\mathbf{F}p\}, \langle \emptyset, \{\mathbf{GF}p\}, \{\mathbf{GF}p\} \rangle) \\
= & E(\{\}, \langle \emptyset, \{\mathbf{GF}p, \mathbf{F}p\}, \{\mathbf{GF}p, \mathbf{F}p\} \rangle) \cup E(\{p\}, \langle \{\}, \{\mathbf{GF}p\}, \{\mathbf{GF}p, \mathbf{F}p\} \rangle) \\
= & E(\{\}, \langle \emptyset, \{\mathbf{GF}p, \mathbf{F}p\}, \{\mathbf{GF}p, \mathbf{F}p\} \rangle) \cup E(\{\}, \langle \{p\}, \{\mathbf{GF}p\}, \{\mathbf{GF}p, \mathbf{F}p, p\} \rangle \\
= & \{\langle \emptyset, \{\mathbf{GF}p, \mathbf{F}p\}, \{\mathbf{GF}p, \mathbf{F}p\} \rangle\} \cup \{\langle \{p\}, \{\mathbf{GF}p\}, \{\mathbf{GF}p, \mathbf{F}p, p\} \rangle\}
\end{array}
$$

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

Example: $\mathbf{GF}p$

- Let $s_1 =_{def} \langle \{p\}, \{\mathbf{GF}p\}, \{\mathbf{GF}p, \mathbf{F}p, p\} \rangle$,
  $s_2 =_{def} \langle \emptyset, \{\mathbf{GF}p, \mathbf{F}p\}, \{\mathbf{GF}p, \mathbf{F}p\} \rangle$,
- $Q = \{s_1, s_2\}$,
- $Q_0 = \{s_1, s_2\}$,
- $T: \quad s_1 \to \{s_1, s_2\},$
  $\quad\quad s_2 \to \{s_1, s_2\}$
- $FT = \langle F_1 \rangle$ where $F_1 = \{s_1\}$.



NBAs of disjunctions of formulas

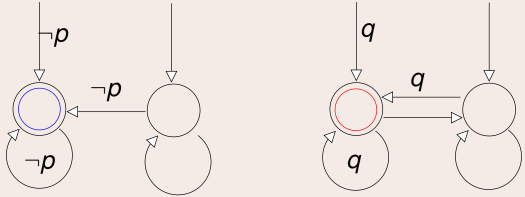
Remark

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
  - $\mathbf{F}p$
  - $\mathbf{G}p$
  - $p\mathbf{R}q$
  - $(\mathbf{GF}p \wedge \mathbf{GF}q) \rightarrow \mathbf{G}r$

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

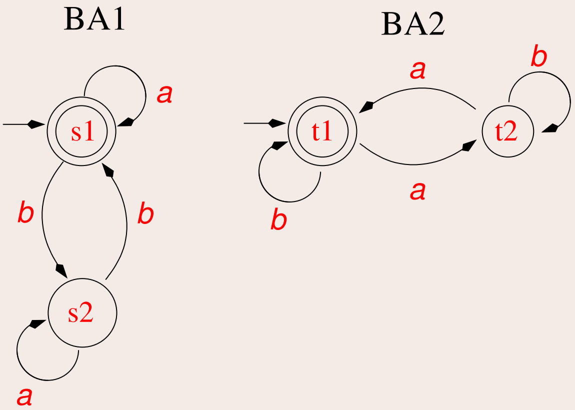
$\Longrightarrow$ the complexity of LTL M.C. grows linearly wrt. the size of the model M and exponentially wrt. the size of the property φ

Final Remarks

- Büchi automata are in general more expressive than LTL!
  $\Longrightarrow$ Some tools (e.g., Spin) allow specifications to be expressed directly as NBAs
  $\Longrightarrow$ complementation of NBA important!
- 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

Ex: Product of Büchi automata

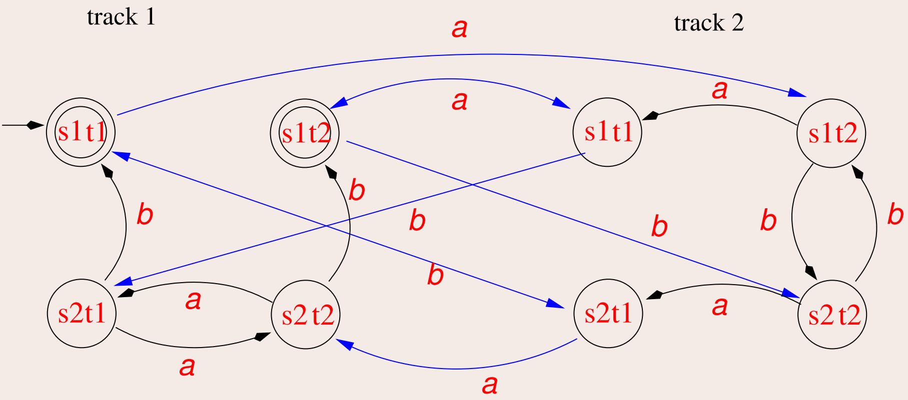
Given the following two Büchi automata (doubly-circled states represent accepting states, *a*, *b* are labels):



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

Ex: Product of Büchi automata

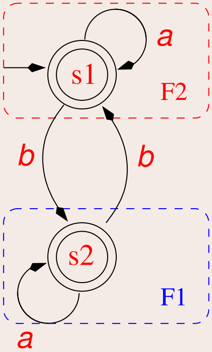
[Solution: The product is:



]

Ex: De-generalization of Büchi Automata

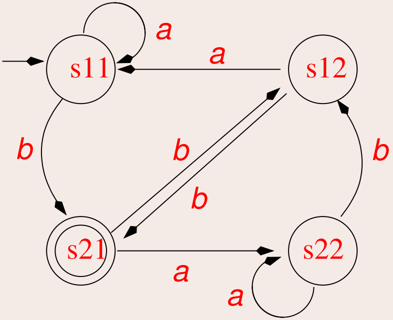
Given the following generalized Büchi automaton $A \stackrel{\text{def}}{=} \langle Q, \Sigma, \delta, I, FT \rangle$, with two sets of accepting states $FT \stackrel{\text{def}}{=} \{F1, F2\}$ s.t. $F1 \stackrel{\text{def}}{=} \{s2\}$, $F2 \stackrel{\text{def}}{=} \{s1\}$:



convert it into an equivalent plain Büchi automaton.

Ex: De-generalization of Büchi Automata

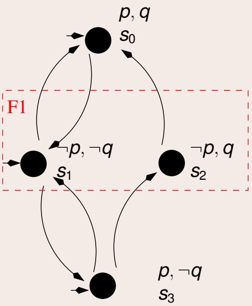
[Solution: The result is:



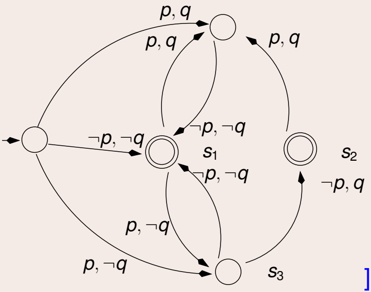
]

Ex: From Kripke models to Büchi automata

Given the following fair Kripke model M, convert it into an equivalent Buchi automaton.



[Solution:



]

Ex: Construction of Büchi Automata

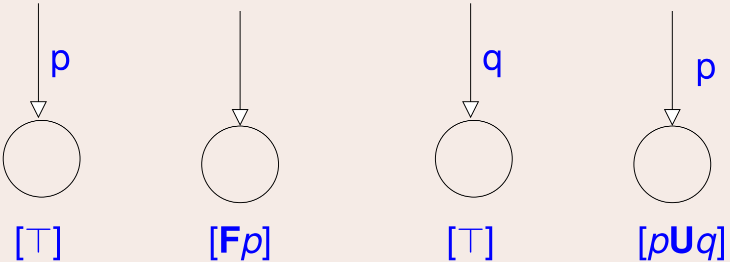
Consider the LTL formula $\varphi \stackrel{\text{def}}{=} (\mathbf{G}\neg p) \rightarrow (p\mathbf{U}q)$.

(a) rewrite φ into Negative Normal Form

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

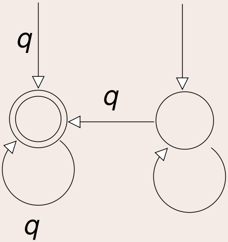
(b) find the initial states of a corresponding Buchi automaton (for each state, define the labels of the incoming arcs and the “next” section.)

[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.

(*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]