Verifica di Sistemi

Automata-based
LTL Model-Checking
Finite state automata

A finite state automaton is a tuple $A = (\Sigma, S, S_0, R, F)$

- $\Sigma$: set of input symbols
- $S$: set of states -- $S_0$: set of initial states ($S_0 \subseteq S$)
- $R: S \times \Sigma \rightarrow 2^S$: the transition relation.
- $F$: set of accepting states ($F \subseteq S$)

A run $r$ on $w=a_1,\ldots,a_n$ is a sequence $s_0,\ldots,s_n$ such that $s_0 \in S_0$ and $s_{i+1} \in R(s_i, a_i)$ for $0 \leq i \leq n$.

A run $r$ is accepting if $s_n \in F$, while a word $w$ is accepted by $A$ if there is an accepting run of $A$ on $w$.

The language $L(A)$ accepted by $A$ is the set of finite words accepted by $A$. 


Finite state automata: union

Given automata $A_1$ and $A_2$, there is an automaton $A$ accepting $L(A) = L(A_1) \cup L(A_2)$.

$A = (\Sigma, S, S_0, R, F)$ is an automaton which just runs non-deterministically either $A_1$ or $A_2$ on the input word.

$S = S_1 \cup S_2$

$F = F_1 \cup F_2$

$S_0 = S_{01} \cup S_{02}$

$R(s,a) = \begin{cases} 
R_1(s,a) & \text{if } s \in S_1 \\
R_2(s,a) & \text{if } s \in S_2 
\end{cases}$
Finite state automata: union

$L(A_1) = (b+ab)^*a$

$L(A_2) = a(a+ba)^*$

$L(A) = L(A_1) \cup L(A_2)$
Finite state automata: intersection

Given automata $A_1$ and $A_2$, there is an automaton $A$ accepting $L(A) = L(A_1) \cap L(A_2)$

$A = (\Sigma, S, S_0, R, F)$ runs simultaneously both automata $A_1$ and $A_2$ on the input word.

$S = S_1 \times S_2$

$F = F_1 \times F_2$

$S_0 = S_{01} \times S_{02}$

$R((s,t),a) = R_1(s,a) \times R_2(t,a)$
Finite state automata: intersection

$L(A_1) = (b + ab)^*a$

$L(A_2) = a (a + ba)^*$

$L(A) = L(A_1) \cap L(A_2)$
Finite state automata: complementation

- If the automaton is deterministic, then it just suffices to set $F^c = S \setminus F$.
- This doesn’t work, though, for non-deterministic automata.
- Solution:
  1. Determinize the automaton using the subset construction.
  2. Complement the resulting deterministic automaton
- The complexity of this process is exponential in the size of the original automaton.
- The number of states of the final automaton is $2^{|S|}$, in the worst case.
Finite state automata: complementation

\[ L(A) = (a + ab)^*a \]

\[ L(A^D) = (a + ab)^*a \]

\[ L(A^c) = \overline{L(A)} \]
**Büchi automata (BA)**

A Büchi automaton is a tuple \( A = (\Sigma, S, S_0, R, F) \)

- \( \Sigma \): set of input symbols
- \( S \): set of states -- \( S_0 \): set of *initial* states ( \( S_0 \subseteq S \) )
- \( R : S \times \Sigma \rightarrow 2^S \): the *transition relation*.
- \( F \): set of *accepting* states ( \( F \subseteq S \) )

- A *run* \( r \) on \( w=a_1, a_2, \ldots \) is an infinite sequence \( s_0, s_1, \ldots \) such that \( s_0 \in S_0 \) and \( s_{i+1} \in R(s_i, a_i) \) for \( i \geq 0 \).

- A *run* \( r \) is *accepting* if some *accepting state in* \( F \) occurs in \( r \) infinitely often.

- A word \( w \) is *accepted* by \( A \) if there is an accepting run of \( A \) on \( w \), and the *language* \( L_\omega(A) \) *accepted* by \( A \) is the set of (infinite) \( \omega \)-*words* accepted by \( A \).
Büchi automata (BA)

A Büchi automaton is a tuple \( A = (\Sigma, S, S_0, R, F) \)

- A \textit{run} \( r \) on \( w = a_1, a_2, \ldots \) is an infinite sequence \( s_0, s_1, \ldots \) such that \( s_0 \in S_0 \) and \( s_{i+1} \in R(s_i, a_i) \) for \( i \geq 0 \).

- Let \( \text{Lim}(r) = \{ s \mid s = s_i \text{ for infinitely many } i \} \)

- A \textit{run} \( r \) is \textit{accepting} if

\[ \text{Lim}(r) \cap F \neq \emptyset \]

- A word \( w \) is \textit{accepted} by \( A \) if there is an accepting run of \( A \) on \( w \).

- The \textit{language} \( L_\omega(A) \) \textit{accepted} by \( A \) is the set of (infinite) \( \omega \)-words accepted by \( A \).
**Büchi automata: union**

Given Büchi automata \(A_1\) and \(A_2\), there is an Büchi automaton \(A\) accepting \(L_\omega(A) = L_\omega(A_1) \cup L_\omega(A_2)\).

The *construction* is the same as for *ordinary automata*.

\(A = (\Sigma, S, S_0, R, F)\) is an automaton which just runs non-deterministically either \(A_1\) or \(A_2\) on the input word.

\[
\begin{align*}
S &= S_1 \cup S_2 \\
F &= F_1 \cup F_2 \\
S_0 &= S_{01} \cup S_{02} \\
R(s,a) &= \begin{cases} 
R_1(s,a) & \text{if } s \in S_1 \\
R_2(s,a) & \text{if } s \in S_2
\end{cases}
\end{align*}
\]
**Büchi automata: intersection**

- The intersection construction for automata does not work for Büchi automata.
- Instead, the intersection for Büchi automata can be defined as follows:

\[ A = (\Sigma S, S_0, R, F) \] intuitively runs simultaneously both automata \[ A_1 = (\Sigma S_1, S_{01}, R_1, F_1) \] and \[ A_2 = (\Sigma S_2, S_{02}, R_2, F_2) \] on the input word.

\[
\begin{align*}
S &= S_1 \times S_2 \times \{1,2\} \\
F &= F_1 \times S_2 \times \{1\} \\
S_0 &= S_{01} \times S_{02} \times \{1\} \\
R((s,t,i),a) &= \begin{cases} 
(s',t',2) & \text{if } s' \in R_1(s,a), \ t' \in R_2(t,a), \ s \in F_1 \text{ and } i=1 \\
(s',t',1) & \text{if } s' \in R_1(s,a), \ t' \in R_2(s,a), \ t \in F_2 \text{ and } i=2 \\
(s',t',i) & \text{if } s' \in R_1(s,a), \ t' \in R_1(t,a) 
\end{cases}
\end{align*}
\]
Büchi automata: intersection

A = (ΣS,S₀,R,F) runs simultaneously both automata A₁ and A₂ on the input word.

\[ S = S₁ \times S₂ \times \{1,2\} \]
\[ F = F₁ \times S₂ \times \{1\} \]
\[ S₀ = S₀₁ \times S₀₂ \times \{1\} \]

\[ R((s,t,i),a) = \begin{cases} 
(s’,t’,2) & \text{if } s’ \in R₁(s,a), t’ \in R₂(t,a), s \in F₁ \text{ and } i=1 \\
(s’,t’,1) & \text{if } s’ \in R₁(s,a), t’ \in R₂(t,a), t \in F₂ \text{ and } i=2 \\
(s’,t’,i) & \text{if } s’ \in R₁(s,a), t’ \in R₁(t,a) 
\end{cases} \]

The automaton remembers 2 tracks, one for each automaton, and points to one of the tracks. As soon as it goes through an accepting state on the current track, it changes track.

The accepting condition and the transition relation ensure that this change of track must happens infinitely often.
Büchi automata: intersection

\[ A = (\Sigma, S, S_0, R, F) \] runs simultaneously both automata \( A_1 \) and \( A_2 \) on the input word.

\[
S = S_1 \times S_2 \times \{1,2\}
\]

\[
F = F_1 \times S_2 \times \{1\}
\]

\[
S_0 = S_{01} \times S_{02} \times \{1\}
\]

\[
R((s,t,i),a) = \begin{cases} 
(s',t',2) & \text{if } s' \in R_1(s,a), t' \in R_2(t,a), s \in F_1 \text{ and } i=1 \\
(s',t',1) & \text{if } s' \in R_1(s,a), t' \in R_2(t,a), t \in F_2 \text{ and } i=2 \\
(s',t',i) & \text{if } s' \in R_1(s,a), t' \in R_1(t,a)
\end{cases}
\]

As soon as it visits an accepting state in track 1, it switches to track 2 and then to track 1 again but only after visiting an accepting state in the track 2.

Therefore, to visit infinitely often a state in \( F \) (\( F_1 \)), the automaton must also visit infinitely often some state of \( F_2 \).
Büchi automata: complementation

It’s a complicated construction -- the standard subset construction for *determinizing automata does not work* as *non-deterministic automata are more powerful* than *deterministic ones* (e.g. $L_\omega=(0+1)^*1^\omega$)

Solution (resorts to another kind of automaton):

- Transform the (non-deterministic) Büchi automaton into a (non-deterministic) *Rabin automaton* (a more general kind of $\omega$-automaton).
- Determinize and then complement the Rabin automaton.
- Transform the Rabin automaton into a Büchi automaton.

• Therefore, also *Büchi automata are closed under complementation.*
**Rabin automata**

- A **Rabin automaton** is like a Büchi automaton, except that the accepting condition is defined differently.
- \( A = (\Sigma, S, S_0, R, F) \), where \( F=\left((G_1,B_1),\ldots,(G_m,B_m)\right) \).
- and the acceptance condition for a run \( r = s_0s_1\ldots \) is as follows: for some \( i \in \{1,\ldots,m\} \)
  - \( \text{Lim}(r) \cap G_i \neq \emptyset \) [a state in \( G_i \) occurs infinitely often] and
  - \( \text{Lim}(r) \cap B_i = \emptyset \) [all states in \( B_i \) occur finitely many times]

in other words, there is a pair \( (G_i,B_i) \) such that the “good” set \( G_i \) is visited *infinitely often*, while the “bad” set \( B_i \) is visited only *finitely many times*. 
Rabin versus Büchi automata

The Büchi automaton for $L_\omega = (0+1)^*1^\omega$

The Rabin automaton for $L_\omega = (0+1)^*1^\omega$

The Rabin automaton has $F = (\{t\}, \{s\})$

Note that the Rabin automaton is deterministic.
Language emptiness for Büchi automata

The *emptiness problem for Büchi automata* is the problem of *deciding* whether the language accepted by a Büchi automaton $A$ is empty, i.e. if $L(A)=\emptyset$.

**Theorem:** The *emptiness problem for Büchi automata* is *decidable in linear time*, i.e. in time $O(|A|)$.

**Fact:** $L(A) = \emptyset$ *iff* in the Büchi automaton there is *no reachable cycle* $A$ containing a state in $F$. 
In other words, $L(A) \neq \emptyset$ iff there is a cycle containing an accepting state, which is also reachable from some initial state of the automaton.

We need to find whether there is such a reachable cycle.

We could simply compute the SCCs of $A$ using the standard DFS algorithm, and check if there exists a reachable (nontrivial) SCC containing a state in $F$.

But this is usually too inefficient in practice. We will therefore use a more efficient nested DFS (more efficient in the average-case).
Efficient language emptiness for BA

Input: A
Initialize: Stack₁ := Ø, Stack₂ := Ø
Table₁ := Ø, Table₂ := Ø

Algorithm Main()
  foreach s ∈ Init
    if s ∉ Table₁ then
      DFS₁(s);
    output("empty");
  return;

Algorithm DFS₁(s)
  push(s,Stack₁);
  hash(s,Table₁);
  foreach t ∈ Succ(s)
    if t ∉ Table₁ then
      DFS₁(t);
  if s ∈ F then
    DFS₂(s,s);
  pop(Stack₁);

Algorithm DFS₂(s,s’)
  push(s,Stack₂);
  hash(s,Table₂);
  foreach t ∈ Succ(s) do
    if t ∉ Table₂ then
      DFS₂(t,s’);
    else if t = s’
      output("not empty");
      output(Stack₁,Stack₂,t);
    return;
  pop(Stack₂);

Note: upon finding a bad cycle, Stack₁+Stack₂+t, determines a counterexample: a bad cycle reached from an init state.
**Generalized Büchi automata (GBA)**

*Generalized Büchi automaton:* $A = (\Sigma, S, S_0, R, (F_0, \ldots, F_{m-1}))$

- A *run* $r$ on $w=a_1a_2\ldots$ is an infinite sequence $s_0s_1\ldots$ such that $s_0 \in S_0$ and $s_{j+1} \in R(s_j, a_j)$ for $j \geq 0$.
- Let $\text{Lim}(r) = \{ s \mid s = s_k \text{ for infinitely many } k \}$
- A run $r$ is *accepting* if for each $0 \leq i < m$
  $$\text{Lim}(r) \cap F_i \neq \emptyset$$

Any *Generalized Büchi automaton* can be easily transformed into a *Büchi automaton* as follows:

$$L(A) = \bigcap_{i \in \{0, \ldots, m-1\}} L(<\Sigma, S, S_0, R, F_i>)$$

This transformation is *not very efficient*, though.
From GBA to BA efficiently

Generalized Büchi automaton: \( A = (\Sigma, S, S_0, R, (F_0, \ldots, F_{m-1})) \)

A generalized Büchi automaton \( A \) can be efficiently transformed into a Büchi automaton \( A' = (\Sigma, S', S'_0, R', F') \) as follows:

\[
\begin{align*}
S' &= S \times \{0, \ldots, m-1\} \\
F' &= F_i \times \{i\} \text{ for some } 0 \leq i < m \\
S'_0 &= S_0 \times \{i\} \text{ for some } 0 \leq i < m
\end{align*}
\]

\[
R'((s, i), a) = \begin{cases} 
(s', (i+1 \mod m)) & \text{if } s' \in R(s, a) \text{ and } s \in F_i \\
(s', i) & \text{if } s' \in R(s, a) \text{ and } s \not\in F_i
\end{cases}
\]

Notice that the transformation above expands the automaton size by a factor of \( m \) (compare with Büchi Intersection).
LTL and Büchi automata: example

• The following Büchi automaton recognizes the models of the LTL formula $p U q$

• Indeed, all these models have the form:
  
  $p^* q \ AP^\omega$

  where by $AP^\omega$ we mean any infinite sequence of atomic propositions in $AP$. 

![Büchi automaton diagram](image-url)
LTL and Büchi automata: example

• The following Büchi automaton recognizes the models of the LTL formula $p \mathcal{U} q$

• Indeed, all these models have the form:
  $$p^* q \text{AP}^\omega$$

where by $\text{AP}^\omega$ we mean any infinite sequence of atomic propositions in $\text{AP}$.

Notice that for convenience, we shall associate *symbols to states* instead of arcs (the general mapping between the two versions of Büchi automata can be easily defined).
LTL-semantics and Büchi automata

• A formula $\psi$ expresses a property of $\omega$-words, i.e., an $\omega$-language $L(\psi) \subseteq \Sigma_{AP}^\omega$.

• For $\omega$-word $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots \in \Sigma_{AP}^\omega$, let $\sigma^i = \sigma_i, \sigma_{i+1}, \sigma_{i+2} \ldots$ be the suffix of $\sigma$ starting at position $i$. We defined the “satisfies” relation, $|= $, inductively:
  
  • $\sigma^i |= p_j$ iff $p_j \in \sigma_i$ (for $p_j \in AP$).
  • $\sigma^i |= \neg \psi$ iff not $\sigma^i |= \psi$.
  • $\sigma^i |= \psi_1 \lor \psi_2$ iff $\sigma^i |= \psi_1$ or $\sigma^i |= \psi_2$.
  • $\sigma^i |= X\psi$ iff $\sigma^{i+1} |= \psi$.
  • $\sigma^i |= \psi_1 U \psi_2$ iff $\exists k \geq i. (\sigma^k |= \psi_2$ and $\forall 0 \leq j < k. \sigma^j |= \psi_1)$
  • $\sigma^i |= \psi_1 R \psi_2$ iff $\forall k \geq i. (\sigma^k |= \psi_2$ or $\exists 0 \leq j < k. \sigma^j |= \psi_1)$

• We can then define the language $L(\psi) = \{ \sigma | \sigma^0 |= \psi \}$. 

Relation with Kripke structures

We extend our definition of “satisfies” to transition systems, or *Kripke structures*, as follows:

• given a run $\pi = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k \rightarrow \ldots$ of $K_{AP}$, let

  $$L(\pi) = L(s_0) L(s_1) \ldots L(s_k) \ldots$$

notice that $L(\pi) \in \Sigma_{AP}^\omega$

• Then $K_{AP} \models \psi$ iff *for all* computations (runs) $\pi$ of $K_{AP}$, $L(\pi) \models \psi$.

In other words:

• setting $L(K_{AP}) = \{L(\pi) \mid \pi \text{ is an infinite path in } K_{AP}\}$

  $$K_{AP} \models \psi \iff L(K_{AP}) \subseteq L(\psi).$$
LTL Model Checking: explanation

\[ K_{AP} \models \psi \iff L(K_{AP}) \subseteq L(\psi) \]
\[ \iff L(K_{AP}) \cap (\Sigma_{AP}^\infty \setminus L(\psi)) = \emptyset \]
\[ \iff L(K_{AP}) \cap L(\neg \psi) = \emptyset \]
\[ \iff L(K_{AP}) \cap L(A\neg \psi) = \emptyset \]
Relation with Kripke structures

We can transform any Kripke structure into a Büchi automaton as follows:

\[ K_{AP} \]

\[ \begin{align*}
\{p,q\} & \quad s_0 \\
\{p\} & \quad s_1 \\
\{q\} & \quad s_2 
\end{align*} \]

\[ A_K \]

\[ \begin{align*}
\{p,q\} & \quad s_0 \\
\{p\} & \quad s_1 \\
\{q\} & \quad s_2 
\end{align*} \]

where every state is accepting!
LTL Model Checking

System Model: $K$

LTL property: $\psi$

Model Checker

Check that $K \models \psi$
by checking that
$L(K) \cap L(A_{\neg \psi}) = \emptyset$

Convert $\neg \psi$ to a Büchi automaton $A_{\neg \psi}$, so that
$L(\neg \psi) = L(A_{\neg \psi})$

Yes!

No! + “counterexample”
The algorithmic tasks to perform

We have reduced LTL model checking to two tasks:

1. Convert an LTL formula $\varphi$ (i.e. $\neg\psi$) into a Büchi automaton $A_{\varphi}$, such that $L(\varphi) = L(A_{\varphi})$.

   • Can we do this in general? .... Yes!!!....... 

2. Check whether $K_{AP} \models \psi$, by checking whether the intersection of languages $L(K_{AP}) \cap L(A_{\neg\psi})$ is empty.

   • It is actually unwise to first construct all of $K_{AP}$, because $K_{AP}$ can be far too big (state explosion).
   • Instead, it is possible to perform the check by constructing states of $K_{AP}$ only as needed.
**LTL to BA translation**

- First, let’s put LTL formulas $\varphi$ in **normal form** where:
  - $\neg$’s have been “**pushed in**”, applying only to **propositions**.
  - the only propositional operators are $\neg$, $\land$, $\lor$.
  - the only temporal operators are $X$, $U$ and its dual $R$.
- We can use the following rules:
  - $p \rightarrow q \equiv \neg p \lor q$ (**definition**);
  - $\neg(p \lor q) \equiv \neg p \land \neg q$ (**De Morgan’s low**);
  - $\neg(p \land q) \equiv \neg p \lor \neg q$ (**De Morgan’s low**);
  - $\neg \neg p \equiv p$ (**double negation low**);
  - $\neg(p \ U q) \equiv (\neg p) \ R (\neg q)$ ;
  - $\neg (p \ R q) \equiv (\neg p) \ U (\neg q)$ ;
  - $F \ p \equiv \top \ U \ p$ ; $G \ p \equiv \bot \ R \ p$ ;
  - $\neg X \ p \equiv X \neg p$ (**linearity**)
LTL to BA translation

• First, let’s put LTL formulas $\varphi$ in **normal form**
  • $\neg$ ‘s have been “pushed in”, applying only to propositions.

• We use the following rules:

  - $p \rightarrow q \equiv \neg p \lor q$ ; $\neg (p \lor q) \equiv \neg p \land \neg q$ ; $\neg (p \land q) \equiv \neg p \lor \neg q$ ; $\neg \neg p \equiv p$ ;
  - $\neg (p \text{ U } q) \equiv (\neg p) \text{ R } (\neg q)$ ; $\neg (p \text{ R } q) \equiv (\neg p) \text{ U } (\neg q)$
  - $F \text{ p } \equiv T \text{ U } p$ ; $G \text{ p } \equiv \bot \text{ R } p$ ; $\neg X \text{ p } \equiv X \neg p$

Examples:

$((p \text{ U } q) \rightarrow F \text{ r}) \equiv (\neg (p \text{ U } q) \lor F \text{ r} \equiv (\neg (p \text{ U } q) \lor (T \text{ U } r) \equiv (\neg p \text{ R } \neg q) \lor (T \text{ U } r))$

GF $p \rightarrow F \text{ r} \equiv (\bot \text{ R } (Fp)) \rightarrow (T \text{ U } p) \equiv (\bot \text{ R } (T \text{ U } p)) \rightarrow (T \text{ U } r) \equiv (\neg (\bot \text{ R } (T \text{ U } p)) \lor (T \text{ U } r) \equiv (T \text{ U } (\bot \text{ R } \neg p)) \lor (T \text{ U } r)$
LTL to BA translation: intuition

- States of $A_\varphi$ will be \textit{sets of subformulas} of $\varphi$, thus if $\varphi = p_1 U \neg p_2$, a state is given by $\Gamma \subseteq \{p_1, p_2, \neg p_2, p_1 U \neg p_2\}$.

- Consider a word $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots \in \Sigma_{AP}^\omega$ such that $\sigma \models \varphi$, where, e.g., $\varphi = \psi_1 U \psi_2$.

- Mark each position $i$ with the set of subformulas $\Gamma_i$ of $\varphi$ that hold true there:

  $\Gamma_0 \quad \Gamma_1 \quad \Gamma_2 \quad \ldots \ldots \ldots$
  $\sigma_0 \quad \sigma_1 \quad \sigma_2 \quad \ldots \ldots \ldots$

- Clearly, $\varphi \in \Gamma_0$. But then, by \textit{consistency}, either:
  - $\psi_1 \in \Gamma_0$ and $\varphi \in \Gamma_1$, or
  - $\psi_2 \in \Gamma_0$.

- The consistency rules dictate our states and transitions.
**LTL to BA translation**

Let $\text{sub}(\varphi)$ denote the set of subformulas of $\varphi$.
We define $A_\varphi = (Q, \Sigma, R, L, \text{Init}, F)$ as follows.

First, the state set:

- $Q = \{ \Gamma \subseteq \text{sub}(\varphi) \mid \text{s.t. } \Gamma \text{ is } \textit{locally consistent} \}$.

- For $\Gamma$ to be *locally consistent* we should have:
  - $\bot \notin \Gamma$
  - if $\psi \lor \gamma \in \Gamma$, then $\psi \in \Gamma$ or $\gamma \in \Gamma$.
  - if $\psi \land \gamma \in \Gamma$, then $\psi \in \Gamma$ and $\gamma \in \Gamma$.
  - if $p_i \in \Gamma$ then $\neg p_i \notin \Gamma$, and if $\neg p_i \in \Gamma$ then $p_i \notin \Gamma$.
  - if $\psi U \gamma \in \Gamma$, then $(\psi \in \Gamma$ or $\gamma \in \Gamma)$.
  - if $\psi R \gamma \in \Gamma$, then $\gamma \in \Gamma$. 


LTL to BA translation

Now, labeling the states of $A_\varphi$:

- The labeling $L: Q \mapsto \Sigma$ is $L(\Gamma) = \{l \in \text{sub}(\varphi) \cap \Sigma \mid l \in \Gamma\}$.

- Now, a word $\sigma = \sigma_0 \sigma_1 \ldots \in (\Sigma_{AP})^\omega$ is in $L(A_\varphi)$ \textit{iff} there is a run $\pi = \Gamma_0 \rightarrow \Gamma_1 \rightarrow \Gamma_2 \rightarrow \ldots$ of $A_\varphi$, s.t., $\forall i \geq 0$, we have that $\sigma_i$ \textit{“satisfies”} $L(\Gamma_i)$, i.e., $\sigma_i$ is a \textit{“satisfying assignment”} for $L(\Gamma_i)$.

- This constitutes a slight redefinition of Büchi automata, where \textit{labeling is on the states} instead of on the edges. This facilitates a much more compact $A_\varphi$. 
**LTL to BA translation**

Now, the transition relation, and the rest of $A_\varphi$.

Based on the following *LTL rules*:

- $(\psi \mathbin{U} \gamma) \equiv \gamma \lor (\psi \land X (\psi \mathbin{U} \gamma))$
- $(\psi \mathbin{R} \gamma) \equiv \gamma \land (\gamma \lor X (\psi \mathbin{R} \gamma)) \equiv (\gamma \land \psi) \lor (\gamma \land X(\psi \mathbin{R} \gamma))$

and on the *semantics of X*, we define:

- $R \subseteq Q \times Q$, where $(\Gamma,\Gamma') \in R$ iff:
  - if $(\psi \mathbin{U} \gamma) \in \Gamma$ then $\gamma \in \Gamma$, or $(\gamma \in \Gamma$ and $(\psi \mathbin{U} \gamma) \in \Gamma'$).
  - if $(\psi \mathbin{R} \gamma) \in \Gamma$ then $\gamma \in \Gamma$, and $(\psi \in \Gamma$ or $(\psi \mathbin{R} \gamma) \in \Gamma'$).
  - if $X \psi \in \Gamma$, then $\psi \in \Gamma'$.
Consider the following formula: \( Fp \equiv \top \cup p \)

\[
\text{sub}(\top \cup p) = \{ \top \cup p, p \}
\]

\[
\text{Init} = \{ \Gamma \in \text{sub}(\top \cup p) \mid \top \cup p \in \Gamma \}
\]
LTL to BA translation: example

Consider the following formula: $\top \mathbf{U} p$

$(\top \mathbf{U} p) \equiv p \lor X (\top \mathbf{U} p)$
Consider the following formula: $\top U p$

$$(\top U p) \equiv p \lor X (\top U p)$$
Consider the following formula: $\top \mathbf{U} p$

$(\top \mathbf{U} p) \equiv p \lor X (\top \mathbf{U} p)$
Consider the following formula: $\tau U p$

$(\tau U p) \equiv p \lor X (\tau U p)$
**LTL to BA translation: example**

In this automaton are runs, e.g. \([\top \U p]^\infty\), where \(p\) never occurs. **These run must not be accepting!**
LTL to BA translation

- Init = \{ \Gamma \in Q \mid \varphi \in \Gamma \}.
- For each \((\psi \mathop{U} \gamma) \in \text{sub}(\varphi)\), there is a set \(F_i \in F\), such that:
  - \(F_i = \{ \Gamma \in Q \mid (\psi \mathop{U} \gamma) \notin \Gamma \) or \(\gamma \in \Gamma \}\)
  - (or equivalently \(F_i = \{ \Gamma \in Q \mid \text{if } (\psi \mathop{U} \gamma) \in \Gamma, \text{then } \gamma \in \Gamma \}\))
  - (notice that if there are no \((\psi \mathop{U} \gamma) \in \text{sub}(\varphi)\), then the acceptance condition is the trivial one: all states are accepting)

Lemma: \(L(\varphi) = L(A_\varphi)\).

But \(A_\varphi\) is now a generalized Büchi automaton …
Consider the following formula: $\top U p$

$$\text{sub}(\top U p) = \{\top U p, p\}$$

$$F = \{F_{\top Up}\} = \{\Gamma \in \text{sub}(\top U p) \mid (\top U p) \notin \Gamma \text{ or } p \in \Gamma\}$$
Consider the following formula: \( G p \equiv \bot R p \)

\[
\text{sub}(\bot R p) = \{ \bot R p , p \}
\]

\[
\text{Init} = \{ \Gamma \in \text{sub}(\bot R p) \mid \bot R p \in \Gamma \}
\]
Consider the following formula: $G\ p \equiv \bot R\ p$

$\text{sub}(\bot R\ p) = \{\bot R\ p, p\}$

$(\bot R\ p) \equiv p \land X (\bot R\ p)$
Consider the following formula: \( G p \equiv \perp R p \)

\[
\text{sub}(\perp R p) = \{\perp R p, p\}
\]

There are no eventualities, hence \( F = \{Q\} \)
Consider the following formula: $p U q$

$\text{sub}(p U q) = \{p U q, p, q\}$

$\text{Init} = \{\Gamma \in \text{sub}(p U p) \mid p U q \in \Gamma\}$
Consider the following formula: $p \bigcup q$

$\text{sub}(p \bigcup q) = \{p \bigcup q, p, q\}$

$\text{Init} = \{\Gamma \in \text{sub}(p \bigcup p) \mid p \bigcup q \in \Gamma\}$
Consider the following formula: $p \mathbin{U} q$

$\text{sub}(p \mathbin{U} q) = \{p \mathbin{U} q, p, q\}$

$(p \mathbin{U} q) \equiv q \lor (p \land X(p \mathbin{U} q))$
Consider the following formula: \( p \lor q \)

\[
\text{sub}(p \lor q) = \{p \lor q, p, q\}
\]

\[
F = \{ F_{p\lor q} \} = \{ \Gamma \in \text{sub}(p \lor q) \mid (p \lor q) \notin \Gamma \lor q \in \Gamma \}
\]
On-the-fly translation algorithm

There is another more efficient way to build the Büchi automaton corresponding to a LTL formula.

• The algorithm proposed by Vardi and his colleagues, is based on the idea of refining states only as needed.

• It only record the necessary information (what must hold) at a state, instead of recording the complete information about each state (both what must hold and what might or might-not hold).

• In a way what “might or might-not hold” is treated as ‘don’t care’ information (which can be filled in, but whose value has no relevant effect).
Algorithm data structure: node

Name: A string identifying the current node.

Father: The name of the father node of current node.

Incoming: List of fully expanded nodes with edges to the current node.

Old: A set of temporal formulae which must hold and in the current node have been processed already.

New: A set of temporal formulae which must hold but in the current node have not been processed yet.

Next: A set of temporal formulae which should hold in the next node (immediate successor) of the current node.

Fully Expanded nodes (i.e. States of the Automaton) are those nodes having the New field empty.
function create graph(\(\phi\))
    return(expand([\(\text{Name} \Leftarrow \text{Father} \Leftarrow \text{new\_name()}\),
        \(\text{Incoming} \Leftarrow \{\text{Init}\}, \text{New} \Leftarrow \{\phi\},
        \(\text{Old} \Leftarrow \emptyset, \text{Next} \Leftarrow \emptyset\}], \emptyset))

function expand (\(\text{Node, Nodes\_Set}\))
    if \(\text{New(\(\text{Node}\))}=\emptyset\) then
        if \(\exists \text{ND} \in \text{Nodes\_Set}\) with \(\text{Old(ND)}=\text{Old(\(\text{Node}\))}\) and
            and \(\text{Next(ND)} = \text{Next(\(\text{Node}\))}\) then
            \(\text{Incoming(ND)} := \text{Incoming(ND)} \cup \text{Incoming(\(\text{Node}\))}\);
            return(\(\text{Nodes\_Set}\));
        else return(expand([\(\text{Name} \Leftarrow \text{Father} \Leftarrow \text{new\_name()}\),
            \(\text{Incoming} \Leftarrow \{\text{Name(\(\text{Node}\))}\},
            \(\text{New} \Leftarrow \text{Next(\(\text{Node}\))}, \text{Old} \Leftarrow \emptyset, \text{Next} \Leftarrow \emptyset\],
            \(\text{Nodes\_Set} \cup \{\text{Node}\}\);)
    else ….

\text{Fully Expanded Nodes}
Example: case of a fully expanded node

Nodes_Set

Name: Node8
Father: Node6
Incoming: 4
New: {}
Next: {⊥ R p}
Old: {⊥ R p ; p}
Example: case of a fully expanded node

**Nodes_Set**

- **Name:** Node9
- **Father:** Node7
- **Incoming:** 4
- **New:** {}
- **Next:** {}
- **Old:** \{p U q ; q\}

**Init**

\[ p U q ; p \]

\[ p U q ; q \]
function `expand (Node, Nodes_Set)`
if `New(Node)` = ∅ then `{preceeding block}`
else

let \( \eta \in \text{New} \);
\( \text{New}(Node) := \text{New}(Node) \setminus \{\eta\} \);

\text{case } \eta \text{ of}

\( \eta = p_i \) or \( \neg p_i \) or \( \top \) or \( \bot \):

\text{if } \eta = \bot \text{ or } \text{Neg}(\eta) \in \text{Old}(Node) \text{ then}
\text{return} (\text{Nodes_Set}) \text{; /* Discard current node */}

\text{else } \text{Old}(Node) := \text{Old}(Node) \cup \{\eta\} \text{;}
\text{return} (\text{expand}(Node, Nodes\ Set)) \text{;}

\( \eta = \mu U \psi \) or \( \mu R \psi \) or \( \mu \lor \psi \) : ....
Splitting a node for Disjunction

Node1

Name: Node1
Father: Node1
Incoming: Init
New: \{p \lor q\}
Next: {} 
Old: {} 

Node2

Name: Node2
Father: Node1
Incoming: Init
New: \{p\}
Next: {} 
Old: \{p \lor q\} 

Node3

Name: Node3
Father: Node1
Incoming: Init
New: \{q\}
Next: {} 
Old: \{p \lor q\}
Splitting a node for Until op.

**Node1**
- **Name**: Node1
- **Father**: Node1
- **Incoming**: Init
- **New**: \{p U q\}
- **Next**: {}
- **Old**: {}

**Node2**
- **Name**: Node2
- **Father**: Node1
- **Incoming**: Init
- **New**: \{p\}
- **Next**: \{p U q\}
- **Old**: \{p U q\}

**Node3**
- **Name**: Node3
- **Father**: Node1
- **Incoming**: Init
- **New**: \{q\}
- **Next**: {}
- **Old**: \{p U q\}
## Splitting a node for Release op.

<table>
<thead>
<tr>
<th>Name</th>
<th>Node1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Father</td>
<td>Node1</td>
</tr>
<tr>
<td>Incoming</td>
<td>Init</td>
</tr>
<tr>
<td>New</td>
<td>{p R q}</td>
</tr>
<tr>
<td>Next</td>
<td>{}</td>
</tr>
<tr>
<td>Old</td>
<td>{}</td>
</tr>
</tbody>
</table>

### Node 2

<table>
<thead>
<tr>
<th>Name</th>
<th>Node2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Father</td>
<td>Node1</td>
</tr>
<tr>
<td>Incoming</td>
<td>Init</td>
</tr>
<tr>
<td>New</td>
<td>{q}</td>
</tr>
<tr>
<td>Next</td>
<td>{p R q}</td>
</tr>
<tr>
<td>Old</td>
<td>{p R q}</td>
</tr>
</tbody>
</table>

### Node 3

<table>
<thead>
<tr>
<th>Name</th>
<th>Node3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Father</td>
<td>Node1</td>
</tr>
<tr>
<td>Incoming</td>
<td>Init</td>
</tr>
<tr>
<td>New</td>
<td>{p,q}</td>
</tr>
<tr>
<td>Next</td>
<td>{}</td>
</tr>
<tr>
<td>Old</td>
<td>{p R q}</td>
</tr>
</tbody>
</table>
**Additional functions**

The function $\text{Neg}()$ is applied only to literals:

$$\text{Neg}(p_i) = \neg p_i \quad \text{Neg}(\top) = \bot$$
$$\text{Neg}(\neg p_i) = p_i \quad \text{Neg}(\bot) = \top$$

The functions $\text{New1}()$, $\text{New2}()$ and $\text{Next1}()$, used for *splitting nodes*, are applied to temporal formulae and defined as follows:

<table>
<thead>
<tr>
<th>$\eta$</th>
<th>$\text{New1}(\eta)$</th>
<th>$\text{Next1}(\eta)$</th>
<th>$\text{New2}(\eta)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu \ U \psi$</td>
<td>${\mu}$</td>
<td>${\mu \ U \psi}$</td>
<td>${\psi}$</td>
</tr>
<tr>
<td>$\mu \ R \psi$</td>
<td>${\psi}$</td>
<td>${\mu \ R \psi}$</td>
<td>${\mu, \psi}$</td>
</tr>
<tr>
<td>$\mu \ \lor \psi$</td>
<td>${\mu}$</td>
<td>$\emptyset$</td>
<td>${\psi}$</td>
</tr>
</tbody>
</table>
function expand (Node, Nodes_Set)
if New(Node) = ∅ then {preceding block}
else
    let η ∈ New;
    New(Node) := New(Node) \ {η};
    case η of
        η = pᵢ or ¬pᵢ or T or ⊥: {preceding block}
η = μ U ψ or μ R ψ or μ ∨ ψ :
    Node1 := [Name ← new_name(), Father ← Name(Node),
               Incoming ← Incoming(Node),
               New ← New(Node) ∪ ({New1(η)} \ Old(Node)),
               Old ← Old(Node) ∪ {η},
               Next ← Next(Node) ∪ {Next1(η)}];
    Node2 := [Name ← new_name(), Father ← Name(Node),
               Incoming ← Incoming(Node),
               New ← New(Node) ∪ ({New2(η)} \ Old(Node)),
               Old ← Old(Node) ∪ {η}, Next ← Next(Node)];
    return (expand(Node2, expand(Node1, Nodes_Set)));
η = μ ∧ ψ : ....

Splitting the node
Splitting the node
function **expand** \((Node, Nodes\_Set)\)

if \( New(Node) = \emptyset \) then \{**preceding block**\}

else

let \( \eta \in New \);

\( New(Node) := New(Node) \setminus \{\eta\} \);

case \( \eta \) of

\( \eta = p_i \) or \( \neg p_i \) or \( T \) or \( \bot \): \{**preceding block**\}

\( \eta = \mu \cup \psi \) or \( \mu \cap \psi \) or \( \mu \vee \psi \): \{**preceding block**\}

\( \eta = \mu \wedge \psi \):

return(\( \text{expand}([\text{Name} \leftarrow \text{Name}(Node), \text{Father} \leftarrow \text{Father}(Node), \text{Incoming} \leftarrow \text{Incoming}(Node), \text{New} \leftarrow (\text{New}(Node) \cup \{\mu, \psi\} \setminus \text{Old}(Node)), \text{Old} \leftarrow \text{Old}(Node) \cup \{\eta\}, \text{Next} = \text{Next}(Node)], Nodes\_Set) \);

\( \eta = X \psi \) : ....
Expanding a node

<table>
<thead>
<tr>
<th>Name</th>
<th>Node1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Father</td>
<td>Node1</td>
</tr>
<tr>
<td>Incoming</td>
<td>Init</td>
</tr>
<tr>
<td>New</td>
<td>{p \land q, \ldots}</td>
</tr>
<tr>
<td>Next</td>
<td>{\ldots}</td>
</tr>
<tr>
<td>Old</td>
<td>{\ldots}</td>
</tr>
</tbody>
</table>

expand

<table>
<thead>
<tr>
<th>Name</th>
<th>Node2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Father</td>
<td>Node1</td>
</tr>
<tr>
<td>Incoming</td>
<td>Init</td>
</tr>
<tr>
<td>New</td>
<td>{p, q, \ldots}</td>
</tr>
<tr>
<td>Next</td>
<td>{\ldots}</td>
</tr>
<tr>
<td>Old</td>
<td>{\ldots, p \land q}</td>
</tr>
</tbody>
</table>
function \texttt{expand} \ (\textit{Node}, \textit{Nodes\_Set})

if \texttt{New(Node)} = \emptyset then \{\textit{preceding block}\}
else

\textit{let } \eta \in \texttt{New};
\texttt{New(Node)} := \texttt{New(Node)} \setminus \{\eta\};

\textit{case } \eta \textit{ of }

\eta = p_i \textit{ or } \neg p_i \textit{ or } \top \textit{ or } \bot: \{\textit{preceding block}\}
\eta = \mu \cup \psi \textit{ or } \mu \mathbin{R} \psi \textit{ or } \mu \lor \psi: \{\textit{preceding block}\}
\eta = \mu \land \psi: \{\textit{preceding block}\}

\eta = X \psi:

\texttt{return(\texttt{expand}(}

\texttt{[Name } \leftarrow \texttt{Name(Node), Father } \leftarrow \texttt{Father(Node),}
\texttt{ Incoming } \leftarrow \texttt{Incoming(Node), New } \leftarrow \texttt{New(Node),}
\texttt{ Old } \leftarrow \texttt{Old(Node)} \cup \{\eta\}, \texttt{ Next } = \texttt{Next(Node)} \cup \{\psi\}],
\texttt{Nodes\_Set});

\texttt{esac;}

end \texttt{expand};
Expanding a node

<table>
<thead>
<tr>
<th>Name</th>
<th>Node1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Father:</td>
<td>Node1</td>
</tr>
<tr>
<td>Incoming:</td>
<td>Init</td>
</tr>
<tr>
<td>New:</td>
<td>{X p,\ldots}</td>
</tr>
<tr>
<td>Next:</td>
<td>{\ldots}</td>
</tr>
<tr>
<td>Old:</td>
<td>{\ldots}</td>
</tr>
</tbody>
</table>

expand

<table>
<thead>
<tr>
<th>Name</th>
<th>Node1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Father:</td>
<td>Node1</td>
</tr>
<tr>
<td>Incoming:</td>
<td>Init</td>
</tr>
<tr>
<td>New:</td>
<td>{\ldots}</td>
</tr>
<tr>
<td>Next:</td>
<td>{\ldots, p}</td>
</tr>
<tr>
<td>Old:</td>
<td>{\ldots, X p}</td>
</tr>
</tbody>
</table>
The need for accepting conditions

• **IMPORTANT**: Remember that not every maximal path \( \pi = s_0 s_1 s_2 \ldots \) in the graph determines a model of the formula: the construction above allows some node to contain \( \mu \cup \psi \) while none of the successor nodes contain \( \psi \).

• This is solved again by imposing the **generalized Büchi acceptance conditions**:
  
  • for each subformula of \( \phi \) of the form \( \mu \cup \psi \), there is a set \( F_\phi \in F \), including the nodes \( s \in Q \), such that either \( \mu \cup \psi \not\in \text{Old}(s) \), or \( \psi \in \text{Old}(s) \).
**Complexity of the construction**

**THEOREM**: For any LTL formula $\phi$ a Büchi automaton $A_\phi$ can be constructed which accepts all and only the $\omega$-sequences satisfying $\phi$.

**THEOREM**: Given a LTL formula $\phi$, the Büchi automaton for $\phi$ whose states are $O(2^{\mid \phi \mid})$ (in the worst-case). $[\mid \phi \mid$ is the number of subformulae of $\phi$].

**THEOREM**: Given a LTL formula $\phi$ and a Kripke structure $K_{sys}$ the, the LTL model checking problem can be solved in time $O(|K_{sys}| \cdot 2^{\mid \phi \mid})$. [actually it is $PSPACE$-complete].
LTL to BA: example

• Consider the following formula:
  \[ G p \]
• where \( p \) is an atomic formula.
• Its \emph{negation-normal form} is
  \[ \bot R p \]
**LTL to BA: example**

Current node is Node 1
Incoming = [Init]
Old = []
New = [\( \perp R p \)]
Next = []

New(node) not empty, removing \( \eta = \perp R p \), node *split* into 2, 3, about to expand them
**LTL to BA: example**

Current node is Node 2
Incoming = [Init]
Old = [⊥ R \( p \)]
New = [\( p \)]
Next = [⊥ R \( p \)]

New(node) not empty, removing \( \eta = p \), node replaced by 4 about to expand them
**LTL to BA: example**

Current node is Node 4
Incoming = [Init]
Old = [\( \bot \textbf{R} p ; p \)]
New = []
Next = [\( \bot \textbf{R} p \)]

New(node) empty, no equivalent nodes. About to add, timeshift and expand.
Current node is Node 5
Incoming = [4]
Old = []
New = [⊥ R p]
Next = []

New(node) not empty, removing \( \eta = \bot R p \), node \textit{split} into 6, 7 about to expand them

\((\bot R p) \equiv (p \land \bot) \lor (p \land X(\bot R p))\)
**LTL to BA: example**

Current node is Node 6  
Incoming = [4]  
Old = [\( \bot R p \)]  
New = [\(p\)]  
Next = [\( \bot R p \)]

New(node) not empty, removing \( \eta = p \), node replaced by 8, about to expand it
LTL to BA: example

Current node is Node 8
Incoming = [4]
Old = [⊥ R p ; p]
New = []
Next = [⊥ R p]

New(node) empty, found equivalent old node in Node_Set (4). Returning it instead.
LTL to BA: example

Current node is Node 7
Incoming = [4]
Old = [⊥ \ R \ p]
New = [⊥ ; p]
Next = []

New(node) not empty, removing η = ⊥, inconsistent node deleted - dead end!
**LTL to BA: example**

*From the split of Node 1*

Current node is Node 3  
Incoming = [Init]  
Old = [⊥ R p]  
New = [⊥ ; p]  
Next = []

New(node) not empty, removing \( \eta = \bot \), inconsistent node deleted - dead end!.
LTL to BA: example

Final graph for $Gp \equiv \bot R p$
Consider the following formula:

\[ p \cup q \]

where \( p \) and \( q \) are atomic formulae.
**LTL to BA: example 2**

Current node is Node 1
Incoming = [Init]
Old = []
New = \([p \lor q]\)
Next = []

New(node) not empty, removing \(\eta = p \lor q\) node *split* into 3, 2, about to expand them

\[(p \lor q) \equiv q \lor (p \land X(p \lor q))\]
**LTL to BA: example 2**

Current node is Node 2
Incoming = [Init]
Old = [p ∪ q]
New = [p]
Next = [p ∪ q]

New(node) not empty, removing η = p node replaced by 4, about to expand them
LTL to BA: example 2

Current node is Node 4
Incoming = [Init]
Old = [p U q ; p]
New = []
Next = [p U q]

New(node) empty, no equivalent nodes. Add, timeshift and expand.
Current node is Node 5
Incoming = [4]
Old = []
New = [\(p \cup q\)]
Next = []

New(node) not empty, removing \(\eta = p \cup q\), node *split* into 6, 7, about to expand.

\[(p \cup q) \equiv q \lor (p \land X(p \cup q))\]
Current node is Node 6
Incoming = \([4]\)
Old = \([p \cup q]\)
New = \([p]\)
Next = \([p \cup q]\)

New(node) not empty, removing \(\eta = p\), node replaced by 8, about to expand it
Current node is Node 8
Incoming = [4]
Old = \([\text{p} \lor \text{q}; \text{p}]\)
New = []
Next = \([\text{p} \lor \text{q}]\)

New(node) empty. Found equivalent old note (4) in Node_Set. Returning it instead.
**LTL to BA: example 2**

From the split of Node 5

Current node is Node 7
Incoming = [4]
Old = \([p \cup q]\)
New = \([q]\)
Next = []

New(node) not empty, removing \(\eta = q\), node replaced by 9, about to expand it
Current node is Node 9
Incoming = [4]
Old = [p U q ; q]
New = []
Next = []

New(node) empty, no equivalent node found. Add timeshift and expand
LTL to BA: example 2

Current node is Node 10
Incoming = [9]
Old = []
New = []
Next = []

New(node) empty, no equivalent node found. Add timeshift and expand
**LTL to BA: example 2**

- Current node is Node 11
- Incoming = [10]
- Old = []
- New = []
- Next = []

New(node) empty. Found equivalent old node in Node_Set (10). Returning it instead.
LTL to BA: example 2

From the split of Node 1

Current node is Node 3
Incoming = [Init]
Old = \([p \lor q]\)
New = \([q]\)
Next = []

New(node) not empty, node replaced by 12, about to expand.
LTL to BA: example 2

Current node is Node 12
Incoming = [Init]
Old = [$p \bigcup q ; q$]
New = []
Next = []

New(node) empty. Found equivalent old node (4) in Node_Set. Returning it instead.
Final graph for $p \mathbf{U} q$
Comparison of the two algorithms

The graphs for \( p \cup q \) obtained from the two algorithms
Notes on the algorithm

• Notice that nodes do not necessarily assign truth value to all atomic propositions (in AP)!

• Indeed the labeling to be associated to a node can be any element of $2^{AP}$ which agrees with the literals (AP or negations of AP) in $\text{Old}(\text{Node})$.

• Let $\text{Pos}(q) = \text{Old}(q) \cap \text{AP}$

• Let $\text{Neg}(q) = \{ \eta \in \text{AP} | \neg \eta \in \text{Old}(q) \}$

$$L(q) = \{ X \subseteq \text{AP} | X \supseteq \text{Pos}(q) \land (X \cap \text{Neg}(q)) = \emptyset \}$$
Notes on the algorithm

\[ L(q) = \{ \{p\}, \{p, q\} \} \]

\[ L(q) = \{ \{q\}, \{p, q\} \} \]

\[ L(q) = \{ \{\}, \{p\}, \{q\}, \{p, q\} \} \]
**Composing** $A_{\text{sys}}$ **and** $A_{\phi}$

- In general what we need to do is to compute the *intersection of the languages* recognized by the two automata $A_{\text{sys}}$ and $A_{\phi}$ and check for emptiness.
- We have already seen *(slide 12)* how this can be done.
- When the *System does not* need to satisfy **FAIRNESS** conditions ($A_{\text{sys}}$ has the trivial acceptance condition, i.e. *all the states are accepting*) there is a more efficient construction...
Efficient composition of $A_{sys}$ and $A_{\phi}$

- When $A_{sys}$ have the trivial acceptance condition, i.e. all the states are accepting, there is a more efficient construction.
- In this case we can just compute:

$$A_{sys} \cap A_{\phi} = \langle \Sigma, S_{sys} \times S_{\phi}, R', S_{0sys} \times S_{0\phi}, S_{sys} \times F_{\phi} \rangle$$

- where

$$(<s,t>,a,<s',t'>) \in R' \text{ iff } (s,a,s') \in R_{sys} \text{ and } (t,a,t') \in R_{\phi}$$
Efficient composition of $A_{sys}$ and $A_\phi$

• Notice that in our case both automata have labels in the states (instead of on the transitions).

• This can be dealt with by simply restricting the set of states of the intersection automaton to those which agree on the labeling on both automata.

• Therefore we define

$$A_{sys} \cap A_\phi = < \Sigma, S', R', (S_{0sys} \times S_0\phi) \cap S', S_{sys} \times F_\phi >$$

• where

$$S' = \{(s,t) \in S_{sys} \times S_\phi \mid L_{sys}(s)_{AP(\phi)} = L_\phi(t)\} \text{ and } \forall (<s,t>,<s',t'>) \in R' \text{ iff } (s,s') \in R_{sys} \text{ and } (t,t') \in R_\phi$$