Tecniche di Specifica e di Verifica

Automata-based
LTL Model-Checking

Finite state automata

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

- S: set of input symbols
- *S*: set of *states* -- S_0 : set of *initial states* (S_0 Í *S*)
- *R:S* ´ S ® 2^S : the *transition relation*.
- *F: set of accepting states* (F Í *S*)
- A *run r* on $w=a_1,\ldots,a_n$ is a sequence $s_0,\ldots,s_n$ such that s_0 Î S_0 and s_{i+1} Î R(s_i,a_i) for 0£i£n.
- A *run r* is *accepting* if s_n Î *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 $\mathcal{L}(A) = \mathcal{L}(A_1)$ È $\mathcal{L}(A_2)$

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

$S = S_1$ È S_2

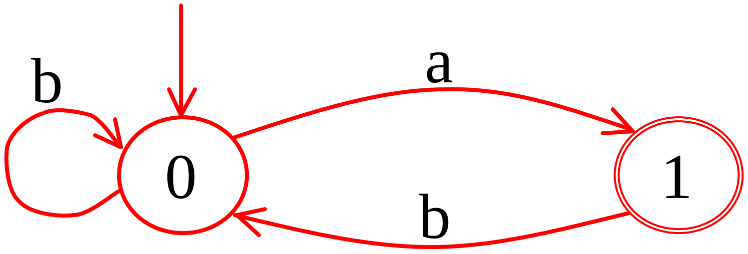
$F = F_1$ È F_2

$S_0 = S_{01}$ È S_{02}

$$R(s,a) = \begin{cases} R_1(s,a) \text{ if } s \text{ Î } S_1 \\ R_2(s,a) \text{ if } s \text{ Î } S_2 \end{cases}$$

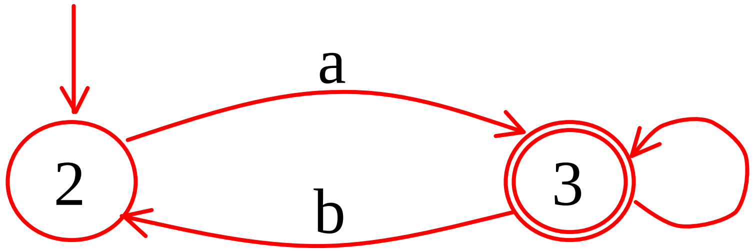
Finite state automata: union

A_1



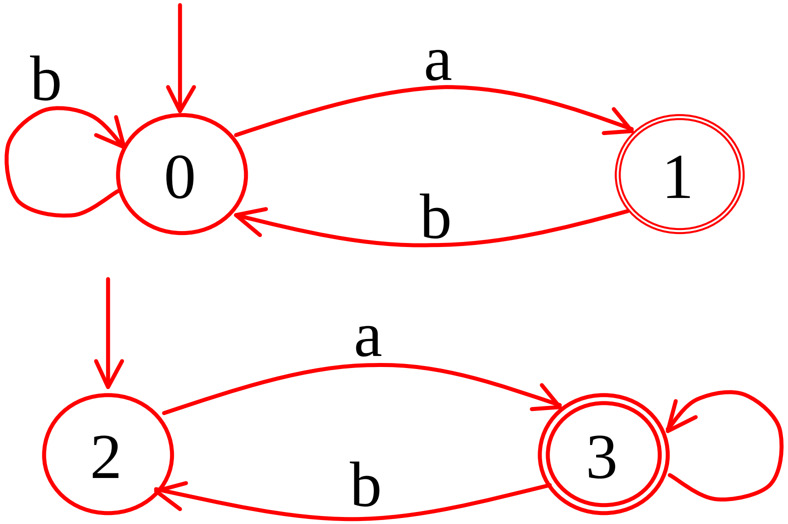
$\mathcal{L}(A_1)=b^*(ab)^*a$

A_2



$\mathcal{L}(A_2)=a(a*ba)^*$

A_1 È A_2



$\mathcal{L}(A)=\mathcal{L}(A_1)$ È $\mathcal{L}(A_2)$

Finite state automata: intersection

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

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

$$S = S_1 \times S_2$$

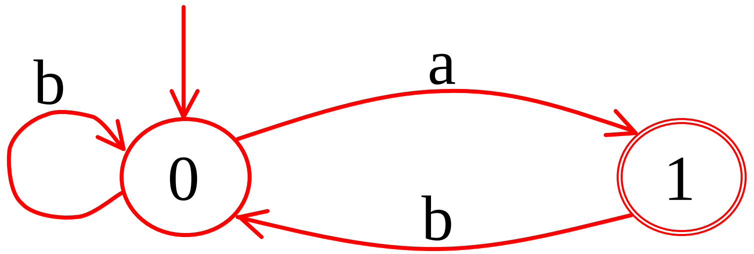
$$F = F_1 \times F_2$$

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

$$R((s,t),a) = R_1(s,a) \times R_2(t,a)$$

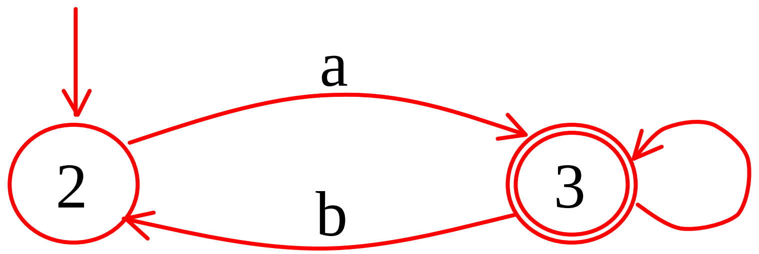
Finite state automata: intersection

A_1



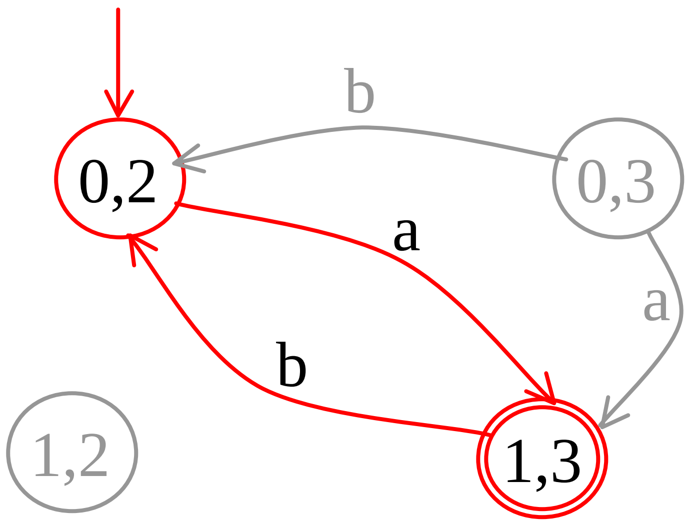
$\mathcal{L}(A_1)=b^*(ab)^*a$

A_2



$\mathcal{L}(A_2)=a(a*ba)^*$

A_1 Ç A_2



$\mathcal{L}(A)=L(A_1)$ Ç $\mathcal{L}(A_2)$

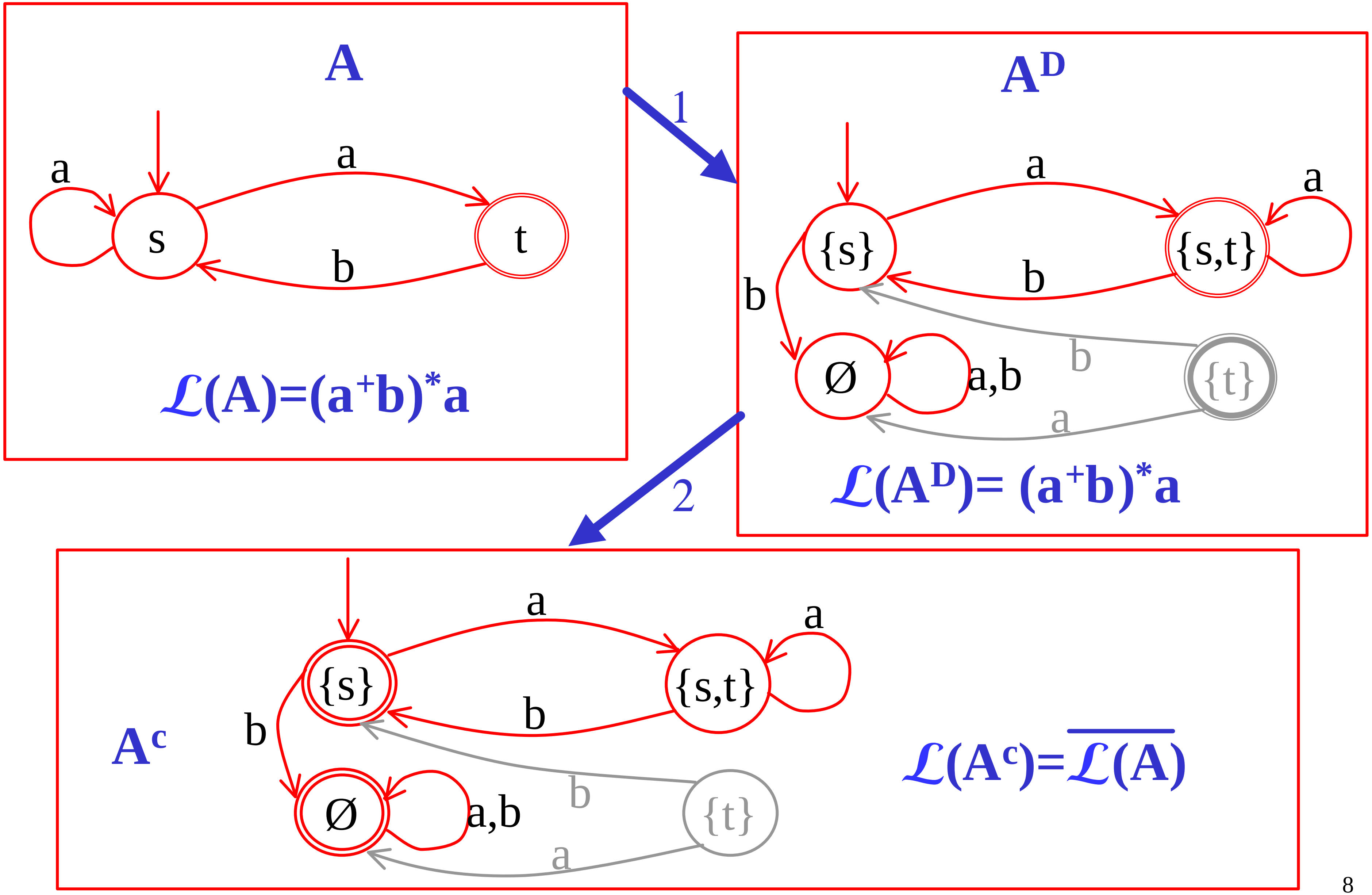
Finite state automata: complementation

- **If the automaton is deterministic, then it just suffices to set $F^c = S - F$.**
- **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



A

$\mathcal{L}(A)=(a^+b)^*a$

A^D

$\mathcal{L}(A^D)= (a^+b)^*a$

A^c

$\mathcal{L}(A^c)=\overline{\mathcal{L}(A)}$

Büchi automata (BA)

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

- Σ: 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 ω is *accepted* by *A* if there is an accepting run of A on ω, and the *language* $\mathcal{L}_\omega(A)$ *accepted* by A is the set of (infinite) ω-words accepted by A.

Büchi automata (BA)

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

- A *run r* on $w=a_1,a_2,\ldots$ is an infinite sequence $s_0,s_1,\ldots$ such that s_0 Î S_0 and s_{i+1} Î R(s_i,a_i) for i ³ 0.
- Let *Lim*(*r*) = { *s* | *s* = s_i for infinitely many *i* }
- A *run r* is *accepting* if

Lim(*r*) Ç *F* ¹ Æ

- A word w is *accepted* by *A* if there is an accepting run of A on w.
- The *language* *ℒ*$_w$*(A)* *accepted* by A is the set of (infinite) w-words accepted by A.

Büchi automata: union

Given Büchi automata A_1 and A_2, there is an Büchi automaton A accepting $\mathcal{L}_\omega(A) = \mathcal{L}_\omega(A_1) \cup \mathcal{L}_\omega(A_2)$.

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.

$S = S_1 \cup S_2$

$F = F_1 \cup F_2$

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

$$R(s,a) = \begin{cases} R_1(s,a) \text{ if } s \in S_1 \\ R_2(s,a) \text{ if } s \in S_2 \end{cases}$$

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.**

$$S = S_1 \times S_2 \times \{1,2\}$$

$$F = F_1 \times S_2 \times \{1\}$$

$$S_0 = S_{01} \times S_{02} \times \{1\}$$

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

Büchi automata: intersection

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

$$S = S_1 \times S_2 \times \{1,2\}$$

$$F = F_1 \times S_2 \times \{1\}$$

$$S_0 = S_{01} \times S_{02} \times \{1\}$$

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

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

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

$$S = S_1 \times S_2 \times \{1,2\}$$

$$F = F_1 \times S_2 \times \{1\}$$

$$S_0 = S_{01} \times S_{02} \times \{1\}$$

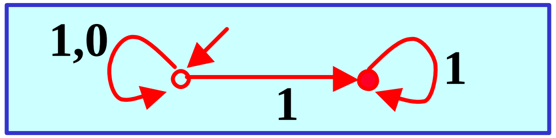
$$R((s,t,i),a) = \begin{cases} (s',t',2) & \text{if } s' \in R_1(s,a),\ t' \in R_2(t,a),\ s \in F_1 \text{ and } i=1 \\ (s',t',1) & \text{if } s' \in R_1(s,a),\ t' \in R_2(t,a),\ t \in F_2 \text{ and } i=2 \\ (s',t',i) & \text{if } s' \in R_1(s,a),\ t' \in R_1(t,a) \end{cases}$$

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 doesn't work* as *non-deterministic automata* are *more powerful* than *deterministic ones* (e.g. $\mathcal{L}_\omega=(0+1)^*1^\omega$)



Solution (resorts to another kind of automaton):

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

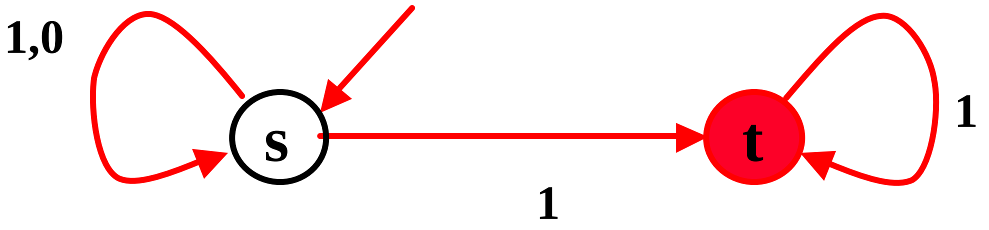
- **Therefore, also *Büchi automata are closed under complementation*.**

Rabin automata

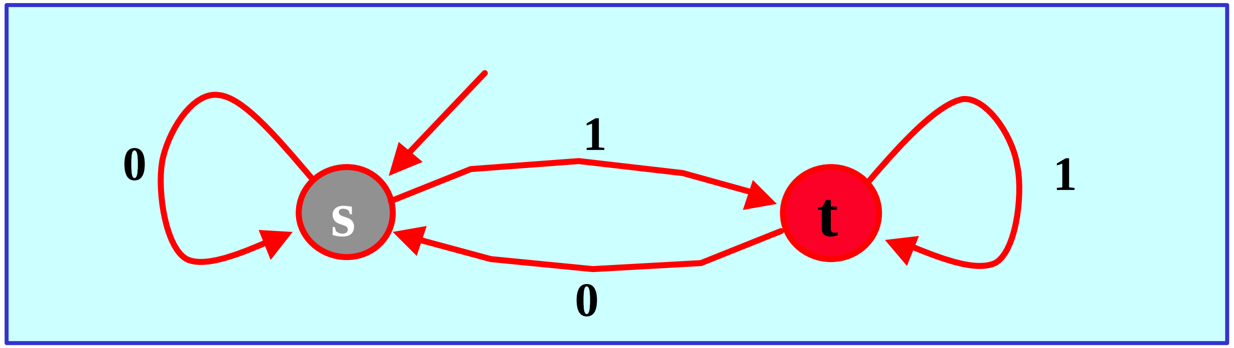
- A Rabin automaton is like a Büchi automaton, except that the accepting condition is defined differently.
- $A = (S,S,S_0,R,F)$, where $F=((G_1,B_1),\dots,(G_m,B_m))$.
- and the acceptance condition for a run $r = s_0,s_1,\dots$ is as follows: for some i
  - Lim(r) Ç G_i ¹ Æ and
  - Lim(r) Ç B_i = Æ

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 often*.

Rabin versus Büchi automata



The Büchi automaton
fot $\mathcal{L}_{w} = (0+1)^{*}1^{w}$



The **Rabin automaton**
fot $\mathcal{L}_{w} = (0+1)^{*}1^{w}$

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 **$\mathcal{L}$(A) = Æ**.

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

Fact: **$\mathcal{L}$(A) = Æ** ***iff*** in the Büchi automaton there is ***no reachable cycle*** **A** ***containing a state in F***.

Language emptiness for Büchi automata

In other words, **$\mathcal{L}$(A)** ¹ Æ ***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: $Stack_1 := \emptyset$, $Stack_2 := \emptyset$
            $Table_1 := \emptyset$, $Table_2 := \emptyset$
Algorithm Main()
    foreach $s \in Init$
        if $s \notin Table_1$ then
            DFS1(s);
    output("empty");
    return;

Algorithm DFS1(s)
    push(s,$Stack_1$);
    hash(s,$Table_1$);
    foreach $t \in Succ(s)$
        if $t \notin Table_1$ then
            DFS1(t);
    if $s \in F$ then
        DFS2(s);
    pop($Stack_1$);
```

```
Algorithm DFS2(s)
    push(s,$Stack_2$);
    hash(s,$Table_2$) ;
    foreach $t \in Succ(s)$ do
        if $t \notin Table_2$ then
            DFS2(t)
        else if t is on $Stack_1$
            output("not empty");
            output($Stack_1$,$Stack_2$,t);
            return;
    pop($Stack_2$);
```

Note: upon finding a bad cycle, $Stack_1$+$Stack_2$+t, determines a counterexample: a bad cycle reached from an init state.

Generalized Büchi automata (GBA)

Generalized Büchi automaton: $A = (\Sigma, S, S_0, R, (F_1, \ldots, F_m))$

- 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 $i \geq 0$.
- Let $Lim(r) = \{ s \mid s = s_i$ for infinitely many $i \}$
- A *run r* is *accepting* if for each $1 \leq i \leq m$

$$Lim(r) \cap F_i \neq \emptyset$$

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

$$\mathcal{L}(\Sigma, S, S_0, R, (F_1, \ldots, F_m)) = \bigcap_{i \in \{1,\ldots,m\}} \mathcal{L}(\Sigma, S, S_0, R, F_i)$$

This transformation is *not very efficient*, though.

From GBA to BA efficiently

Generalized Büchi automaton: $A = (\Sigma, S, S_0, R, (F_1, \ldots, F_m))$

A *Generalized Büchi automaton* can be *efficiently* transformed into a *Büchi automaton* as follows:

$$S' = S \times \{1,\ldots,m\}$$

$F' = F_j \times \{i\}$ for some $1 \le i \le m$

$S'_0 = S_0 \times \{i\}$ for some $1 \le i \le m$

$$R((s,i),a) = \begin{cases} (s', (i \bmod m)+1) & \text{if } s' \in R(s,a) \text{ and } s \in F_i \\ (s',i) & \text{if } s' \in R(s,a) \text{ and } s \notin F_i \end{cases}$$

Notice that the transformation above expands the automaton size by a factor of *m* (compare with *Büchi Intersection*).

LTL-semantics and Büchi automata

- **We can interpret a formula ψ as expressing a property of ω-words, i.e., an ω-language $L(\psi) \subseteq \Sigma_{AP}^{\omega}$.**
- **For ω-word $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots\ldots \in \Sigma_{AP}^{\omega}$, let $\sigma^i = \sigma_i, \sigma_{i+1}, \sigma_{i+2}, \ldots.$ be the suffix of σ starting at position i. We defined the "*satisfies*" relation, $\models$, inductively:**
  - $\sigma \models p_j$ **iff** $p_j \in \sigma_0$ **(for any $p_j \in P$).**
  - $\sigma \models \neg\psi$ **iff** **not** $\sigma \models \psi$.
  - $\sigma \models \psi_1 \vee \psi_2$ **iff** $\sigma \models \psi_1$ **or** $\sigma \models \psi_2$.
  - $\sigma \models X\psi$ **iff** $\sigma^1 \models \psi$.
  - $\sigma \models \psi_1 U \psi_2$ **iff** $\exists i \geq 0$ **such that** $\sigma^i \models \psi_2$, **and** $\forall j, 0 \leq j < i, \sigma^j \models \psi_1$.
- **We can then define the language $\mathcal{L}(\psi) = \{ \sigma \mid \sigma \models \psi \}$.**

Relation with Kripke structures

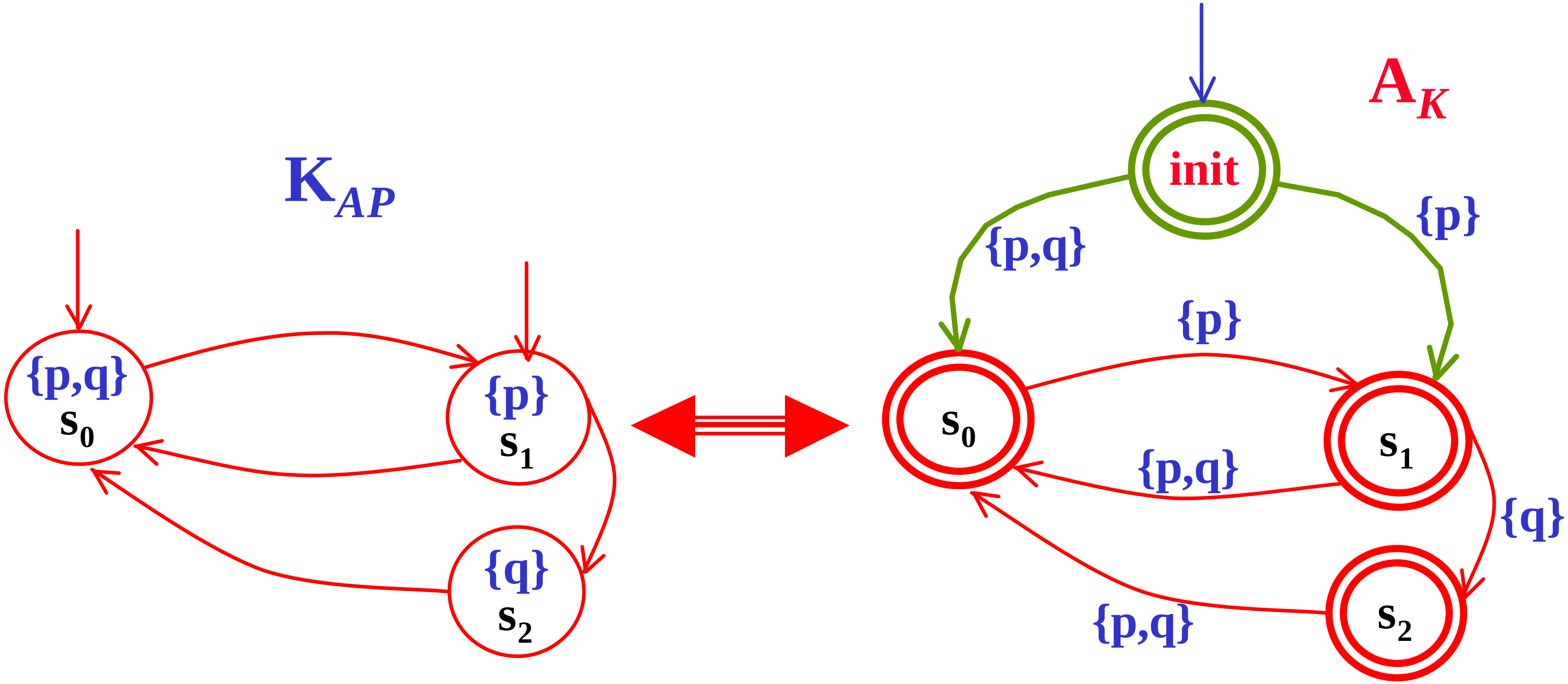
We extend our definition of "*satisfies*" to transition systems, or *Kripke structures*, as follows:

- $\mathbf{K}_{AP} \models \mathrm{y}$ **iff** ***for all*** **computations (runs)** p **of** $\mathbf{K}_{AP}$**,** $\mathcal{L}(\mathrm{p}) \models \mathrm{y}$ **, or in other words, iff**

$$\mathcal{L}(\mathbf{K}_{AP}) \text{ Í } \mathcal{L}(\mathrm{y}).$$

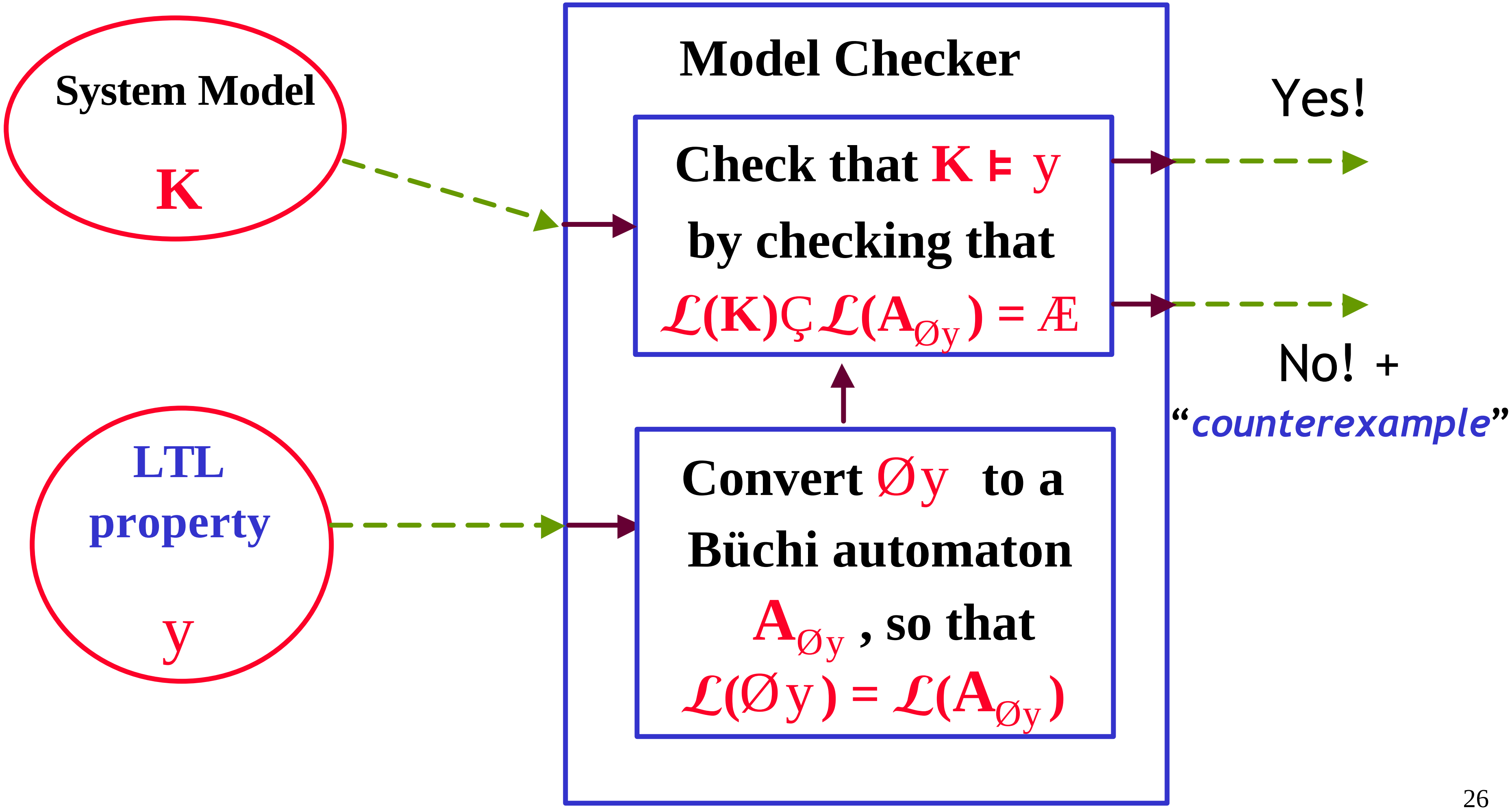
Relation with Kripke structures

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



where *every state is accepting*!

LTL Model Checking



System Model

K

LTL property

y

Model Checker

Check that K ⊨ y by checking that

$\mathcal{L}$(**K**)Ç$\mathcal{L}$(**A**$_{Øy}$) = Æ

Convert Øy to a Büchi automaton A$_{Øy}$ **, so that**

$\mathcal{L}$(Øy) = $\mathcal{L}$(**A**$_{Øy}$)

Yes!

No! +

“*counterexample*”

LTL Model Checking: explanation

$$\begin{aligned}
\mathbf{M} \models \text{y} \quad & \text{Û} \quad \mathcal{L}(\mathbf{K}_{AP}) \ \text{Í} \ \mathcal{L}(\text{y}) \\
& \text{Û} \quad \mathcal{L}(\mathbf{K}_{AP}) \ \text{Ç} \ (\text{S}_{AP}{}^{\text{w}} \setminus \mathcal{L}(\text{y})) = \text{Æ} \\
& \text{Û} \quad \mathcal{L}(\mathbf{K}_{AP}) \ \text{Ç} \ \mathcal{L}(\text{Øy}) = \text{Æ} \\
& \text{Û} \quad \mathcal{L}(\mathbf{K}_{AP}) \ \text{Ç} \ \mathcal{L}(\mathbf{A}_{\text{Øy}}) = \text{Æ} \\
& \text{Û} \quad \mathcal{L}(\mathbf{K}_{AP} \ \text{Ç} \ \mathbf{A}_{\text{Øy}}) = \text{Æ}
\end{aligned}$$

The algorithmic tasks to perform

We have reduced LTL *model checking* to two tasks:

1 **Convert an LTL formula j (i.e. Øy) into a Büchi automaton A_j , such that $\mathcal{L}$(j) = $\mathcal{L}$(A_j).**

- **Can we do this in general? …. Yes!!!……**

2 **Check whether $K_{AP} \models$ y, by checking whether the intersection of languages $\mathcal{L}$(K_{AP}) Ç $\mathcal{L}$($A_{Øy}$) 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 perform the check by *constructing* states of K_{AP} only *as needed*.**

LTL to BA translation

- First, let's put LTL formulas φ in ***normal form*** where:
  - Ø 's have been "**pushed in**", applying only to propositions.
  - the only propositional operators are Ø, Ù,Ú.
  - the only temporal operators are **X**, **U** and its dual **R**.
- In order to do that we use the following rules:
  - p ® q º Ø p Ú q ; p « q º (Ø p Ú q) Ù (Ø q Ú p)
  - Ø(p Ú q) º Ø p Ù Ø q ; Ø(p Ù q) º Ø p Ú Ø q ; Ø Ø p º p
  - Ø(p **U** q) º (Ø p) **R** (Ø q) ; Ø (p **R** q) º (Ø p) **U** (Ø q)
  - F p º ⊤ **U** p ; G p º ⊥ **R** p ; Ø X p º X Ø p

LTL to BA translation

- First, let’s put LTL formulas φ in ***normal form***
  - Ø ‘s have been “**pushed in**”, applying only to propositions.
- We use the following rules:
  - p ® q º Ø p Ú q ; p « q º (Ø p Ú q) Ù (Ø q Ú p)
  - Ø(p Ú q) º Ø p Ù Ø q ; Ø(p Ù q) º Ø p Ú Ø q ; Ø Ø p º p
  - Ø (p **U** q) º (Ø p) **R** (Ø q) ; Ø (p **R** q) º (Ø p) **U** (Ø q)
  - F p º ⊤ **U** p ; G p º ⊥ **R** p ; Ø X p º X Ø p

Examples:

((p U q) ® F r) º Ø(p U q) Ú F r º Ø(p U q) Ú (⊤ U r) º

º (Ø p R Ø q) Ú (⊤ U r)

GF p ® F r º (⊥ **R** (Fp)) ® (⊤ U p) º (⊥ **R** (⊤ U p)) ® (⊤ U r) º

º Ø (⊥ **R** (⊤ U p)) Ú (⊤ U r) º (⊤ U Ø (⊤ U p)) Ú (⊤ U r) º

º (⊤ U (⊥ R Ø p)) Ú (⊤ U r)

LTL to BA translation

- States of $\mathbf{A}_j$ will be ***sets of subformulas*** of j, thus if we have $j = \mathbf{p}_1\mathbf{U}\text{Ø}\mathbf{p}_2$, a state is given by $G \text{ Í } \{\mathbf{p}_1,\text{Ø}\mathbf{p}_2,\mathbf{p}_1\mathbf{U}\text{Ø}\mathbf{p}_2\}$.
- Consider a word $s = s_0, s_1, s_2, \ldots \text{ Î } S_{AP}^{w}$ such that $s \models \varphi$, where, e.g., $j = y_1 \mathbf{U} y_2$.
- Mark each position i with the set of subformulas Γ_i of φ that hold true there:

$$G_0 \quad G_1 \quad G_2 \quad \ldots\ldots\ldots\ldots$$

$$s_0 \quad s_1 \quad s_2 \quad \ldots\ldots\ldots\ldots$$

- Clearly, $j \text{ Î } G_0$. But then, by ***consistency***, either:
  - $y_1 \text{ Î } G_0$ and $j \text{ Î } G_1$, or
  - $y_2 \text{ Î } G_0$.
- The consistency rules dictate our states and transitions.

LTL to BA translation

Let **sub(φ)** denote the set of subformulas of φ.

We define $\mathbf{A}_{\varphi}$ **= (Q, Σ, R, L, Init, F)** as follows.

First, the ***set of states*** of $\mathbf{A}_{\varphi}$ is defined as follows:

- **Q = {**$\Gamma \subseteq$ **sub(φ)** | s.t. Γ is ***locally consistent*** **}**.
  - For Γ to be ***locally consistent*** we should, e.g., have:
    - $\bot \notin \Gamma$
    - if $\psi \vee \gamma \in \Gamma$, then $\psi \in \Gamma$ or $\gamma \in \Gamma$.
    - if $\psi \wedge \gamma \in \Gamma$, then $\psi \in \Gamma$ and $\gamma \in \Gamma$.
    - if $\mathbf{p}_i \in \Gamma$ then $\neg \mathbf{p}_i \notin \Gamma$, and if $\neg \mathbf{p}_i \in \Gamma$ then $\mathbf{p}_i \notin \Gamma$.
    - if $\psi\ \mathbf{U}\ \gamma \in \Gamma$, then ($\psi \in \Gamma$ or $\gamma \in \Gamma$).
    - if $\psi\ \mathbf{R}\ \gamma \in \Gamma$, then $\gamma \in \Gamma$.

LTL to BA translation

Now, ***labeling*** of the states of **A**$_j$ is defined as:

- The labeling **L:Q**↦S is **L(**G**)={ l | l** Î G Ç S **}**.
  - We want a word s = s_0 s_1 **…** Î **(**S_{AP}**)**^{w} to be in ℒ(**A**$_j$) ***iff*** there is a run p = G_0 ® G_1 ® G_2 ® **…** of **A**$_j$ s.t. "**i** Î ℕ, we have that s_i "***satisfies***" **L(**G_i**)**, i.e., s_i is a "***satisfying assignment***" for **L(**G_i**)** .
  - This constitutes a ***slight redefinition of Büchi automata***, where ***labeling is on the states*** instead of on the edges. This facilitates a much more compact **A**$_j$.

LTL to BA translation

Then the ***transition relation*** of **A**$_j$.

It is based on the following ***LTL rules***:

- **(**y **U** g) º g Ú (y Ù **X** **(**y **U** g))
- **(**y **R** g) º g Ù (y Ú **X** **(**y **R** g)) º (g Ù y) Ú (g Ù **X(**y **R** g))

and on the ***semantics*** of the operator **X**.

- **R** Í **Q** ´ **Q** , where **(**G**,**G’**)** Î **R** iff:
  - if **(**y **U** g) Î G then g Î G, or (y Î G and **(**y **U** g) Î G’).
  - if **(**y **R** g) Î G then g Î G, and (y Î G or **(**y **R** g) Î G’).
  - if **X** y Î G, then y Î G’.

LTL to BA translation

- The ***initial states*** of $\mathbf{A}_\varphi$ are $\mathbf{Init} = \{\Gamma \in \mathbf{Q} \mid \varphi \in \Gamma\}$.
- The ***accepting states*** of $\mathbf{A}_\varphi$ are defined as follows:

  for each $(\psi\,\mathbf{U}\gamma) \in \mathbf{sub}(\varphi)$, there is a set $\mathbf{F}_i \in \mathbf{F}$, such that:

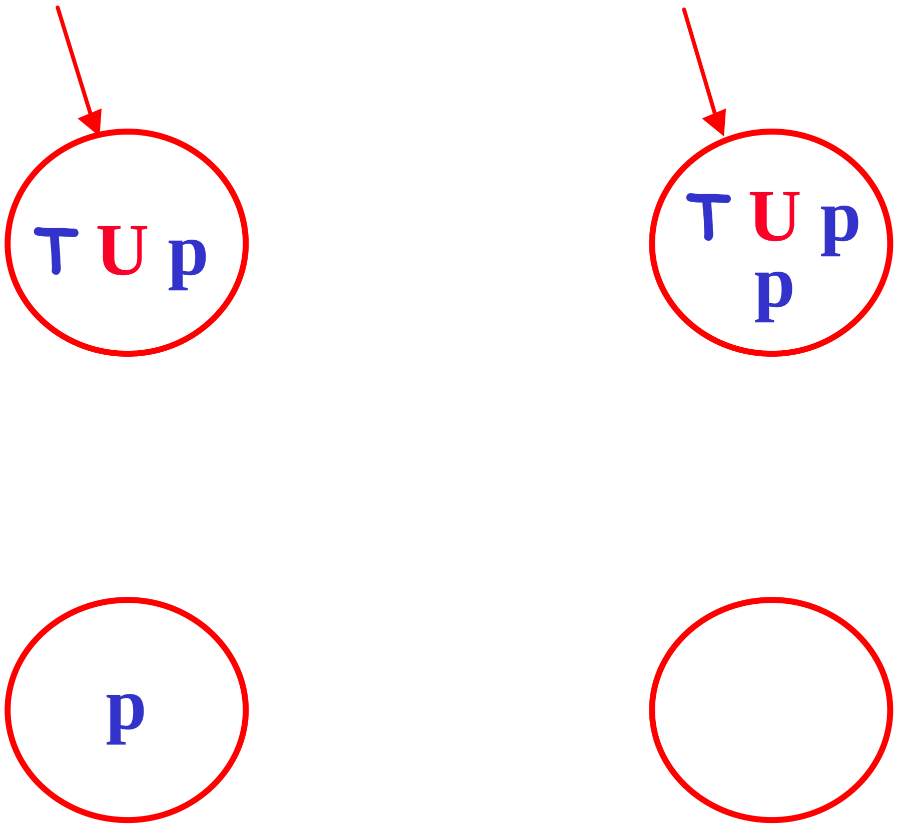
  - $\mathbf{F}_i = \{\Gamma \in \mathbf{Q} \mid (\psi\ \mathbf{U}\ \gamma) \notin \Gamma \text{ or } \gamma \in \Gamma\}$

    or equivalently $\mathbf{F}_i = \{\Gamma \in \mathbf{Q} \mid \text{if } (\psi\ \mathbf{U}\ \gamma) \in \Gamma, \text{ then } \gamma \in \Gamma\}$
  - Notice that if there is ***no*** $(\psi\ \mathbf{U}\ \gamma) \in \mathbf{sub}(\varphi)$, then the acceptance condition is the ***trivial acceptance condition***: i.e., ***all states are accepting***

Lemma: $\mathcal{L}(\varphi) = \mathcal{L}(\mathbf{A}_\varphi)$.

But $\mathbf{A}_\varphi$ is now a ***generalized Büchi automaton*** …

LTL to BA translation: example

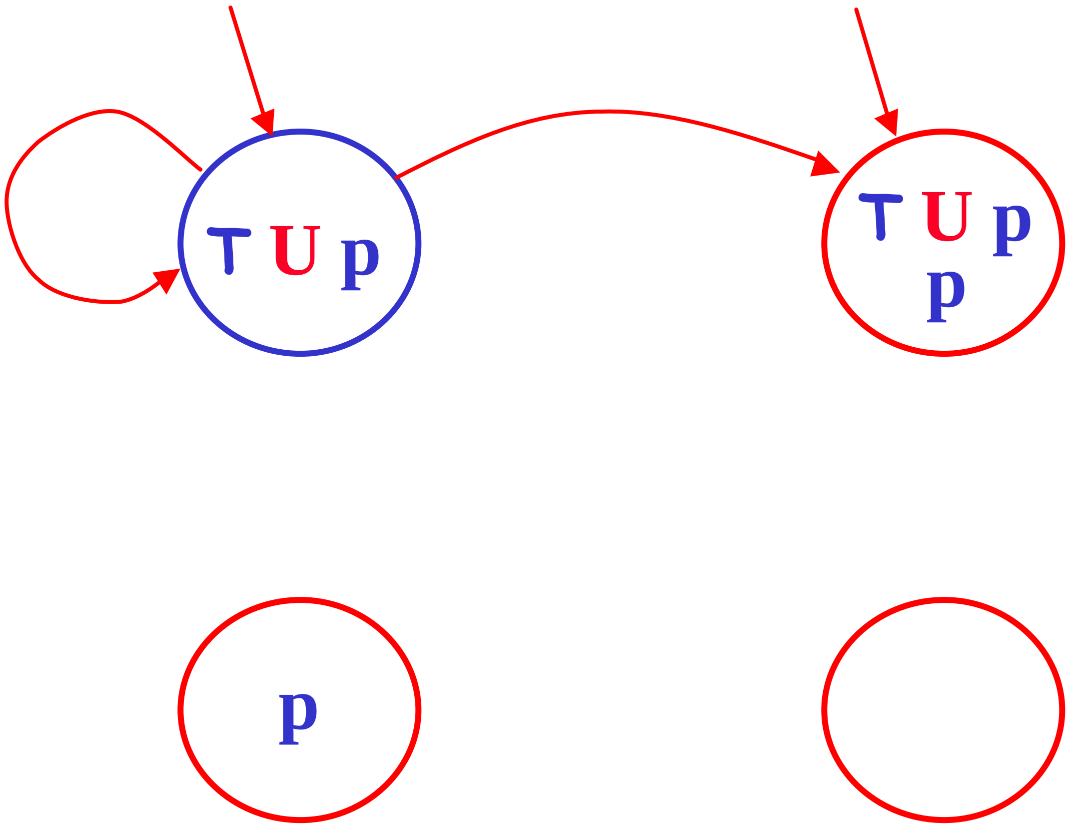


Consider the following formula: $F\,p \equiv \top\,U\,p$

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

$\text{Init} = \{\Gamma \in \text{sub}(\top\,U\,p) \mid \top\,U\,p \in \Gamma\}$

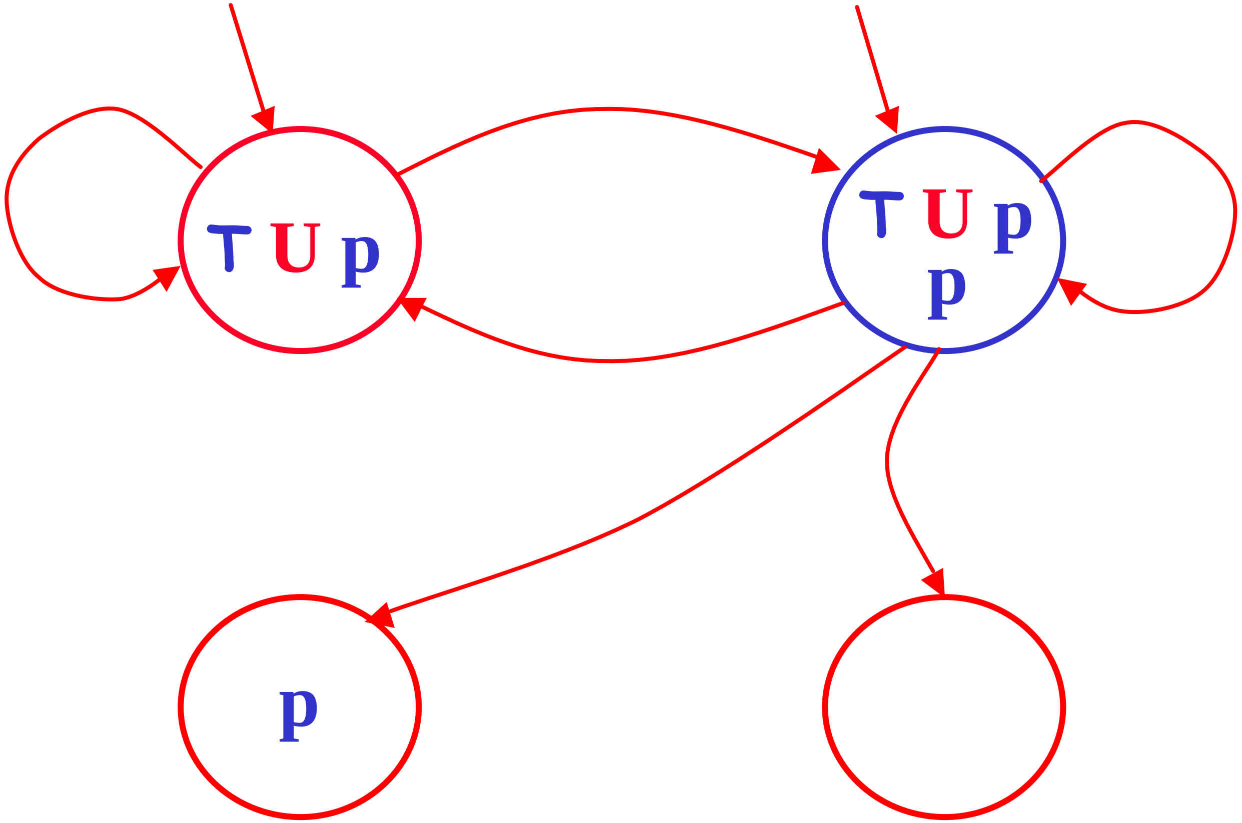
LTL to BA translation: example



Consider the following formula: $\top \, U \, p$

$(\top \, U \, p) \equiv p \vee X\,(\top \, U \, p)$

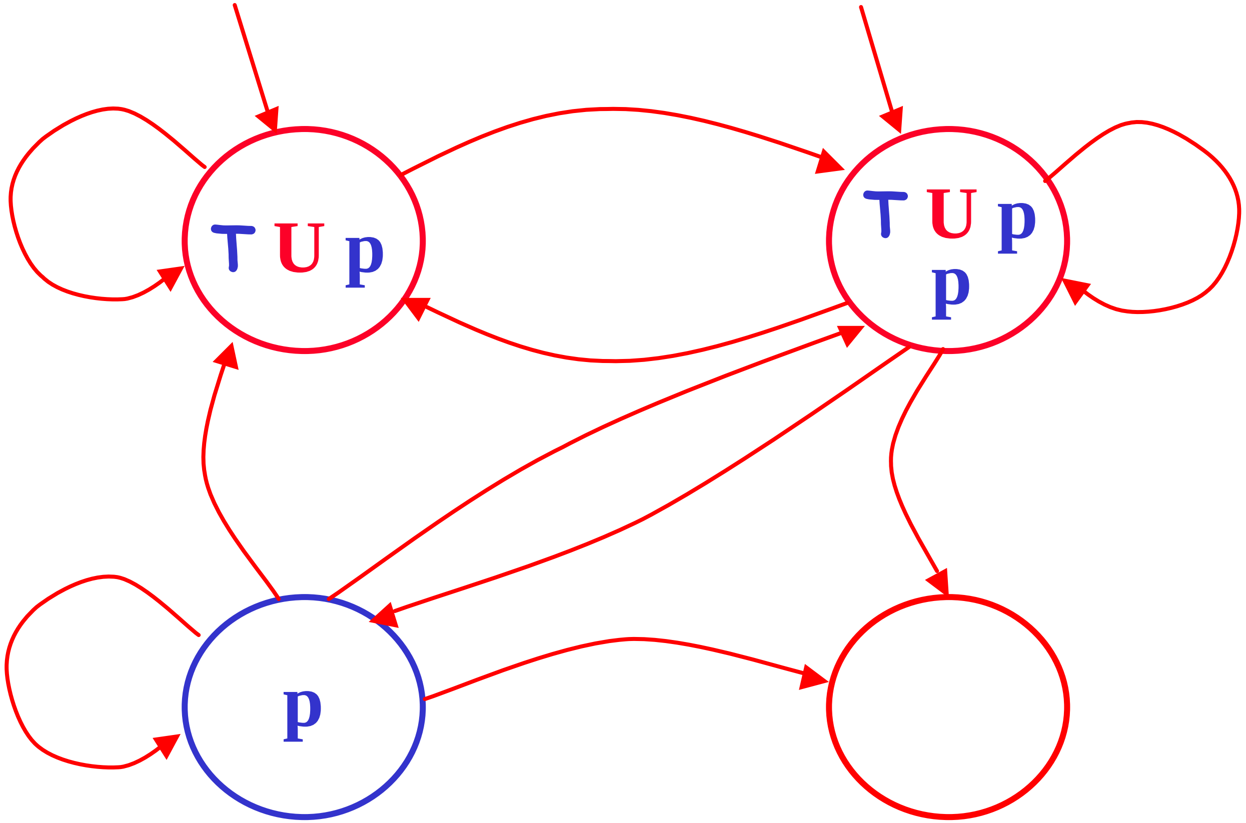
LTL to BA translation: example



Consider the following formula: ⊤ U *p*

(⊤ U *p*) º *p* Ú X (⊤ U *p*)

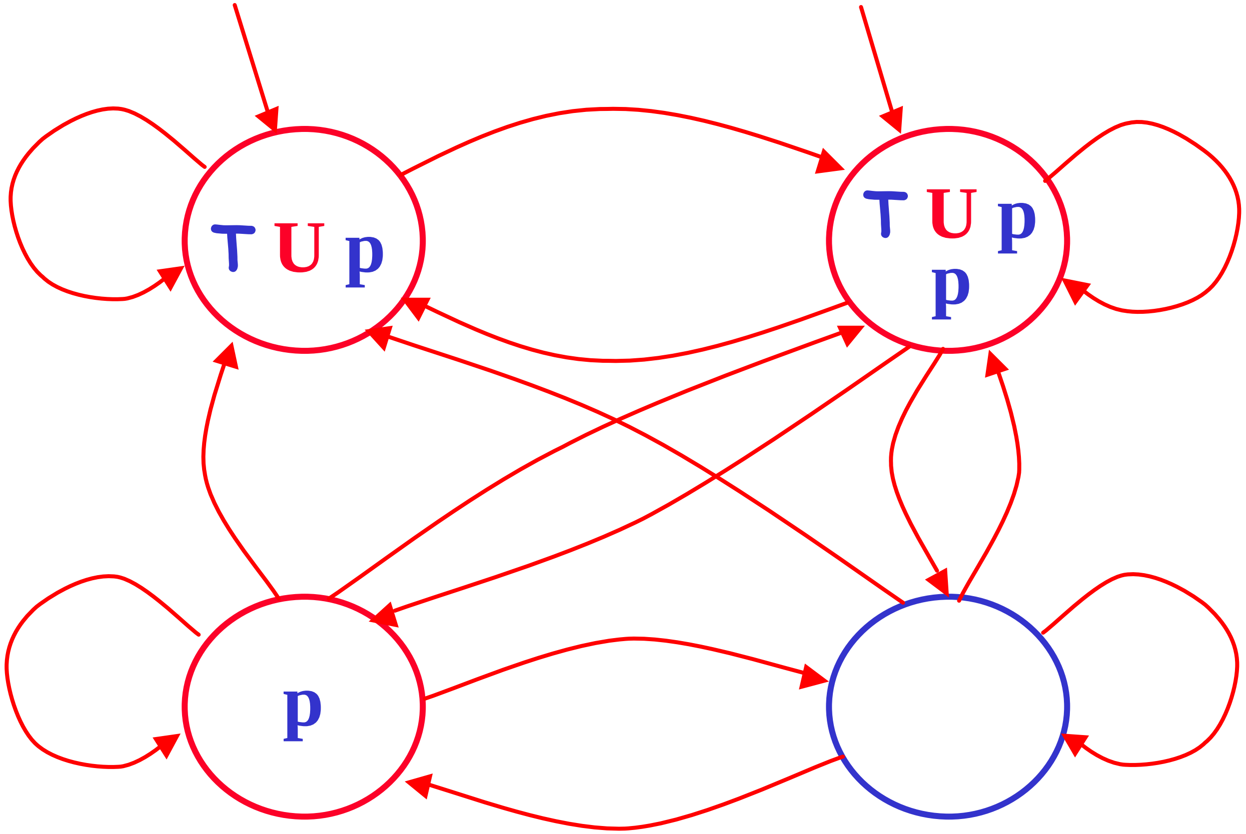
LTL to BA translation: example



Consider the following formula: ⊤ U *p*

(⊤ U *p*) º *p* Ú X (⊤ U *p*)

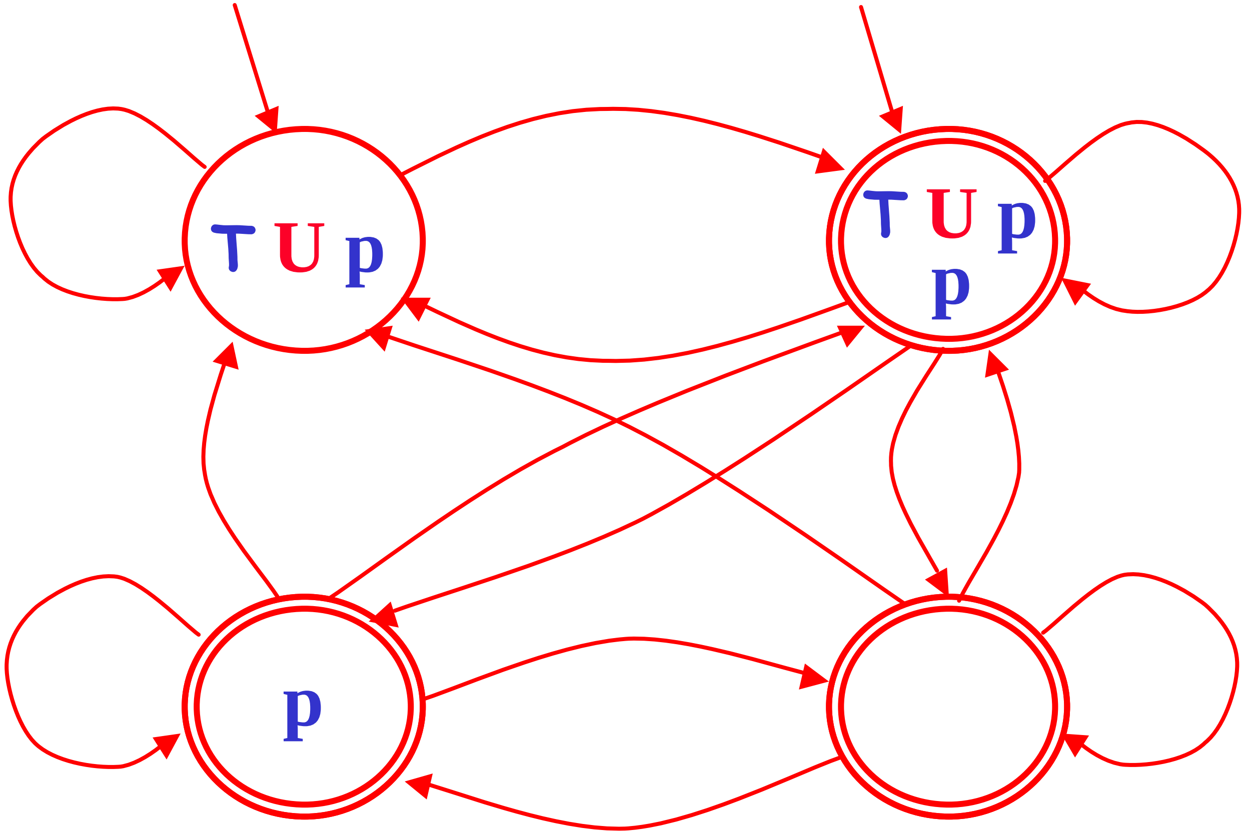
LTL to BA translation: example



Consider the following formula: ⊤ U *p*

(⊤ U *p*) º *p* Ú X (⊤ U *p*)

LTL to BA translation: example

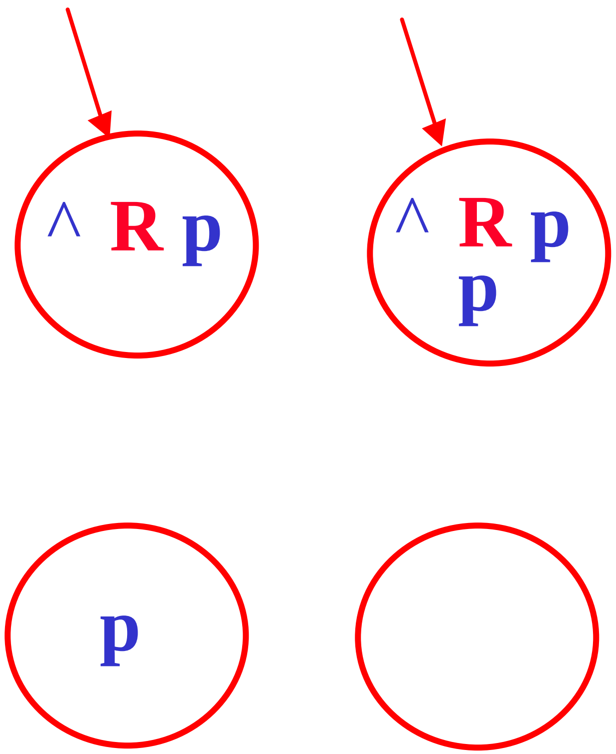


Consider the following formula: $\top \, \mathrm{U} \, p$

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

$$\mathrm{F} = \{\mathrm{F}_{\top \mathrm{U} p}\} = \{G \in \mathrm{sub}(\top \, \mathrm{U} \, p) \mid (\top \, \mathrm{U} \, p) \notin G \text{ or } p \in G\}$$

LTL to BA translation: example

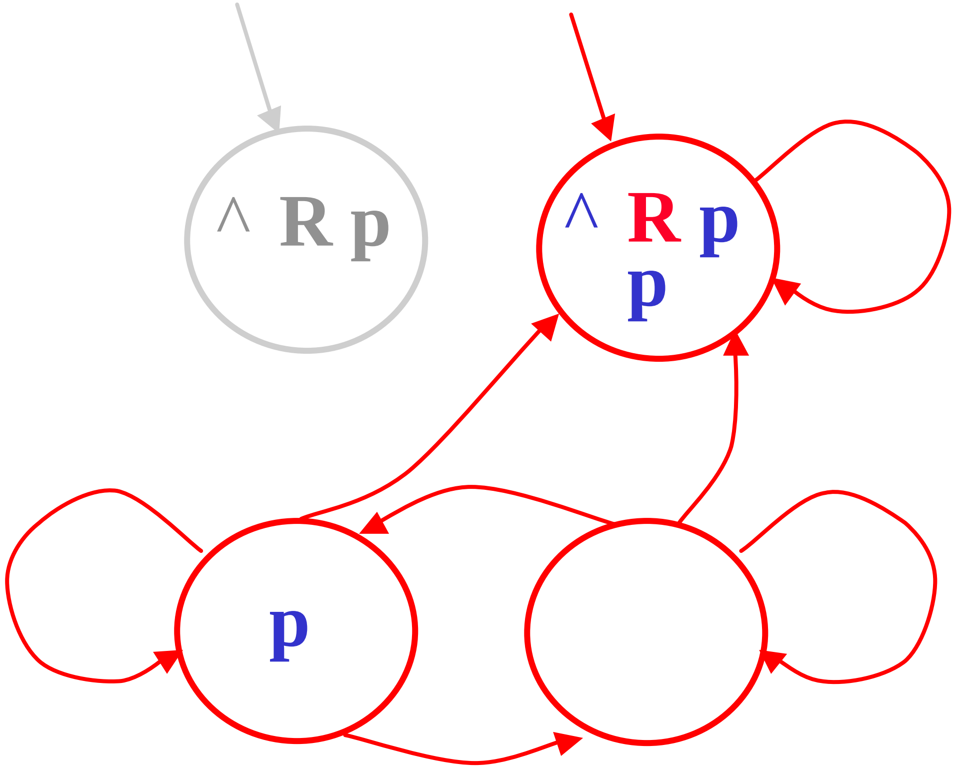


Consider the following formula: G *p* º ∧ R *p*

sub(∧ R *p*) = {∧ R *p* , *p*}

Init = {G Î sub(∧ R *p*) | ∧ R *p* Î G}

LTL to BA translation: example

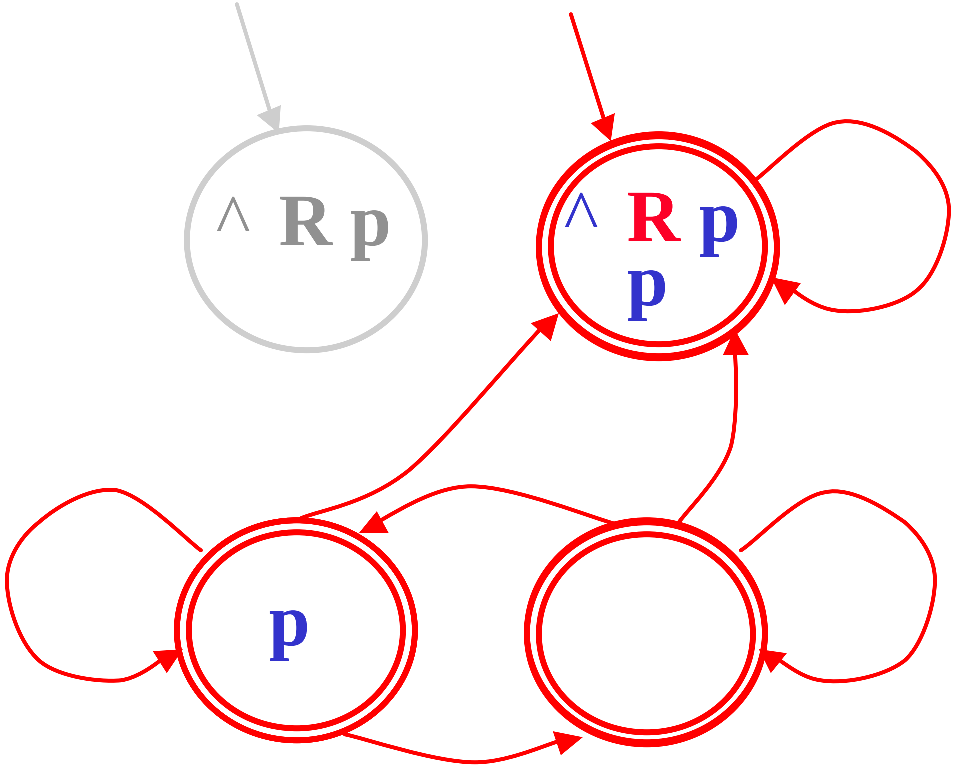


Consider the following formula: G *p* ° ^ R *p*

sub(^ R *p*) = {^ R *p* , *p*}

(^ R *p*) ° *p* Ù X (^ R *p*)

LTL to BA translation: example



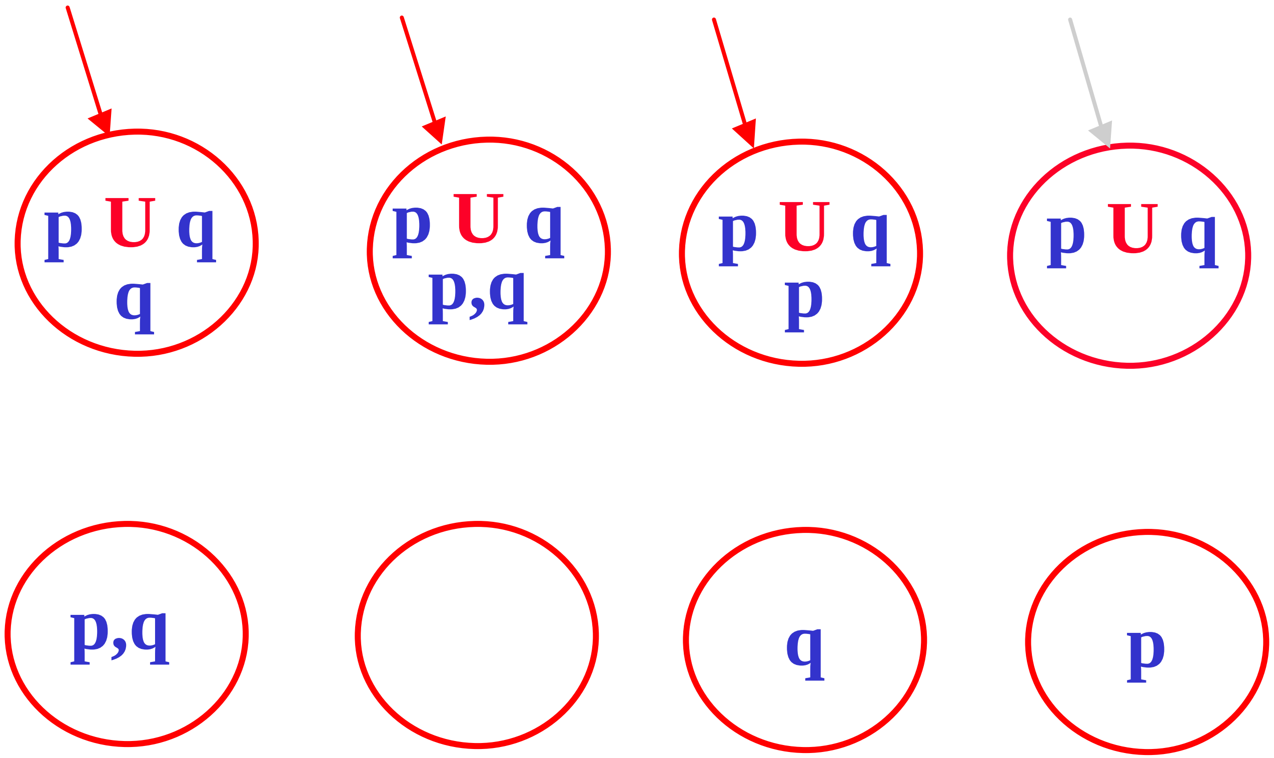
The *trivial acceptance condition*

Consider the following formula: G *p* º ∧ R *p*

sub(∧ R *p*) ={∧ R *p* , *p*}

There are *no eventualities*, hence F = { Q }

LTL to BA translation: example

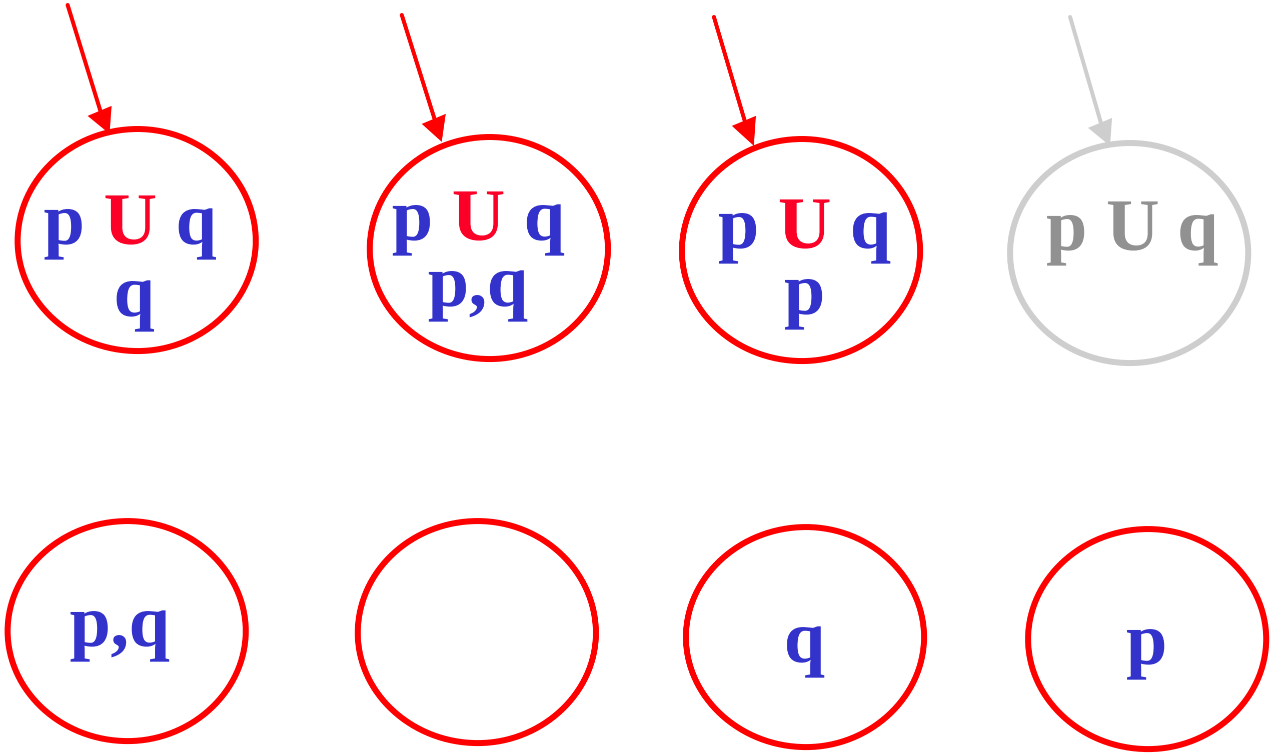


Consider the following formula: ***p* U *q***

sub(*p* U *q*) = {*p* U *q* , *p* , *q*}

Init = {G Î sub(*p* U *p*) | *p* U *q* Î G}

LTL to BA translation: example

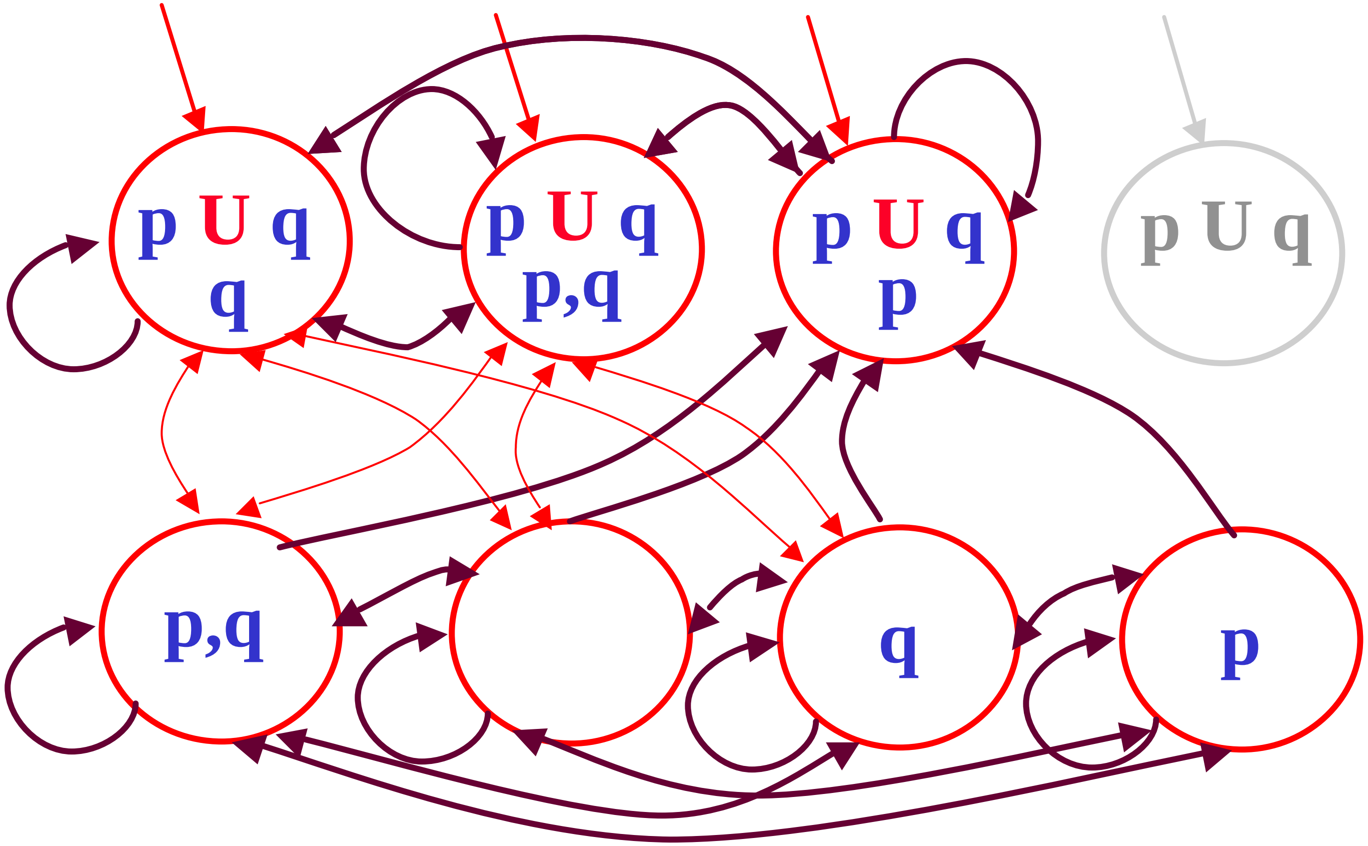


Consider the following formula: ***p*** **U** ***q***

sub(*p* U *q*) = {*p* U *q* , *p* , *q*}

Init = {G Î **sub(*p* U *p*) | *p* U *q*** Î G**}**

LTL to BA translation: example

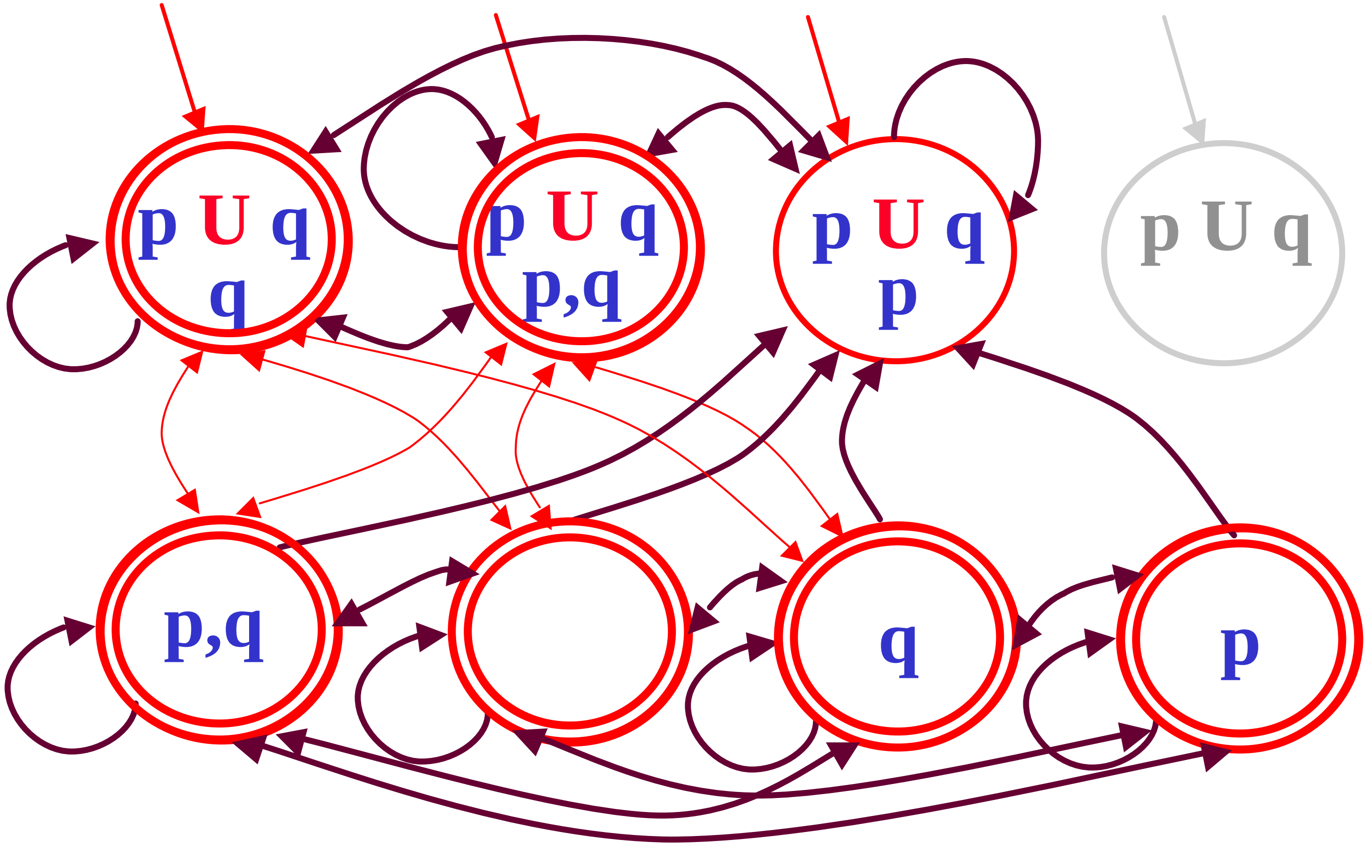


Consider the following formula: ***p* U *q***

sub(*p* U *q*) = {*p* U *q* , *p* , *q*}

(*p* U *q*) º *q* Ú (*p* Ù X (*p* U *q*))

LTL to BA translation: example



Consider the following formula: $p \,\mathrm{U}\, q$

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

$$F = \{\, F_{p\mathrm{U}q} \,\} = \{G \in \mathrm{sub}(p \,\mathrm{U}\, q) \mid (p \,\mathrm{U}\, q) \notin G \text{ or } q \in G\}$$

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*.**

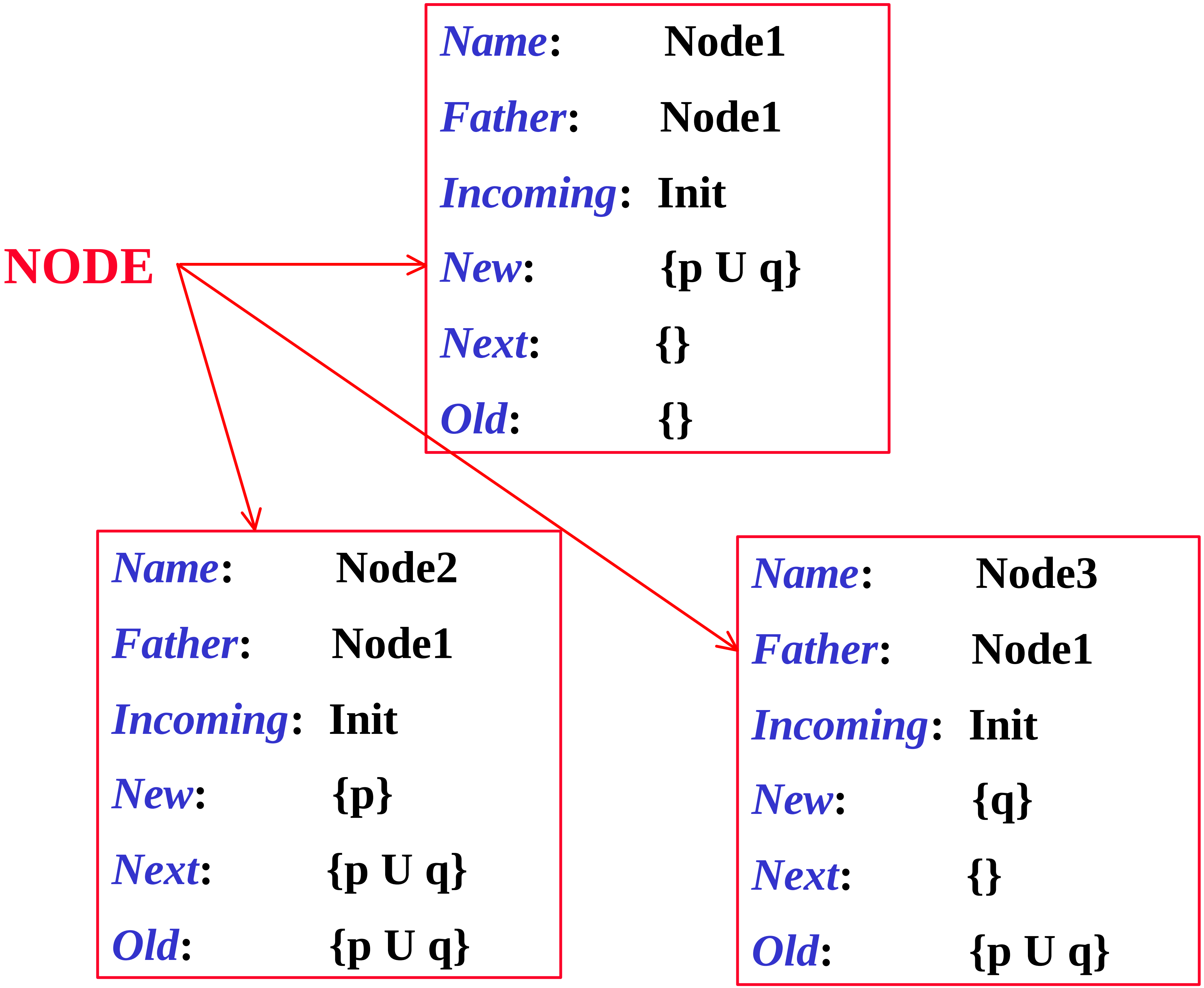
***Father*: The name of the *father node* of *current node*.**

***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*.**



NODE

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

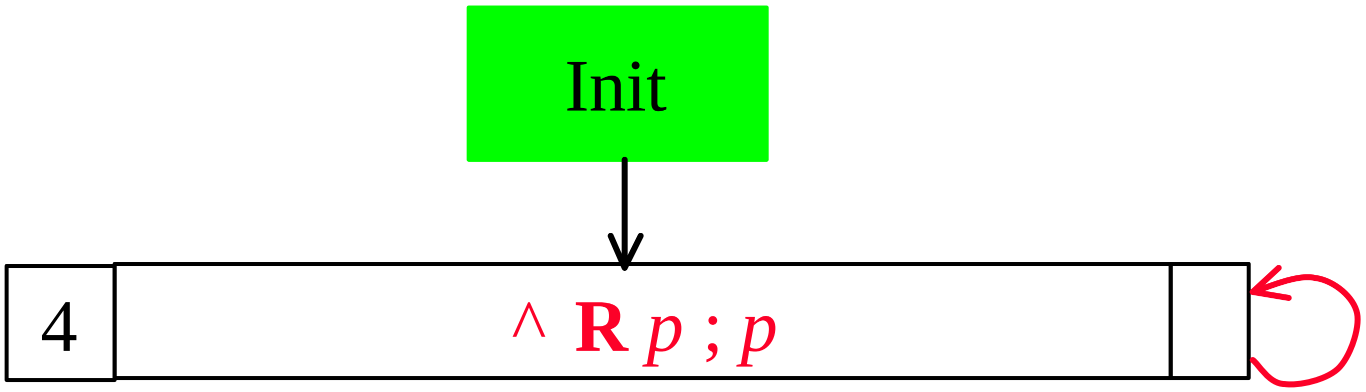
Name: **Node2**
Father: **Node1**
Incoming: **Init**
New: **{p}**
Next: **{p U q}**
Old: **{p U q}**

Name: **Node3**
Father: **Node1**
Incoming: **Init**
New: **{q}**
Next: **{}**
Old: **{p U q}**

```
function create_graph(f)
    return(expand([Name⇐Father⇐new_name(),
                   Incoming⇐{Init}, New⇐{f},
                   Old⇐Æ, Next⇐Æ], Æ)
```

```
function expand (Node, Nodes_Set)
if New(Node) = Æ then
    if $ ND Î Nodes_Set with (Old(ND)=Old(Node) and
                              Next(ND) = Next(Node)) then
        Incoming(ND) ⇐ Incoming(ND) ∪ Incoming(Node);
        return(Nodes_Set);
    else return(expand([Name ⇐ Father ⇐ new_name(),
                        Incoming ⇐ {Name(Node)},
                        New ⇐ Next(Node), Old ⇐ Æ, Next ⇐ Æ],
                       Nodes_Set È {Node});
else ….
```

Nodes_Set



Name**:** **Node8**

Father**:** **Node6**

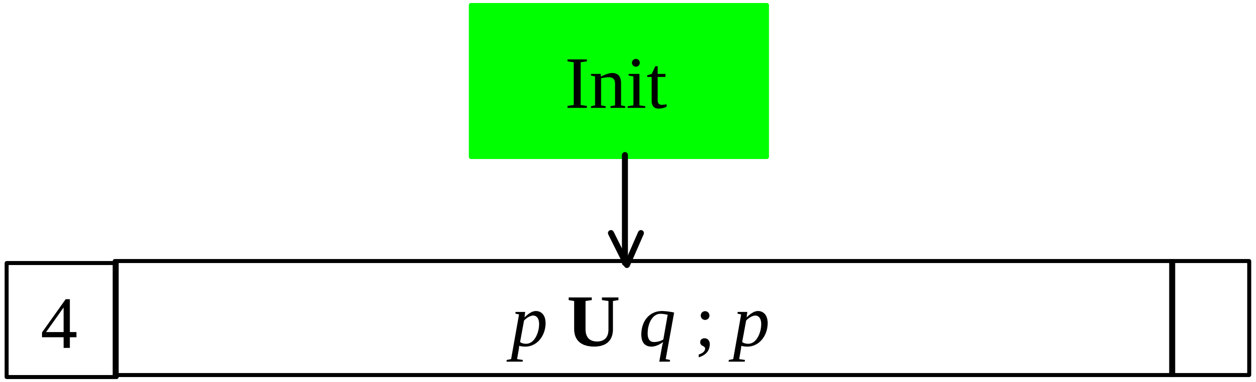
Incoming**:** **4**

New**:** **{}**

Next**:** **{**∧ **R** *p***}**

Old**:** **{**∧ **R *p* ; *p*}**

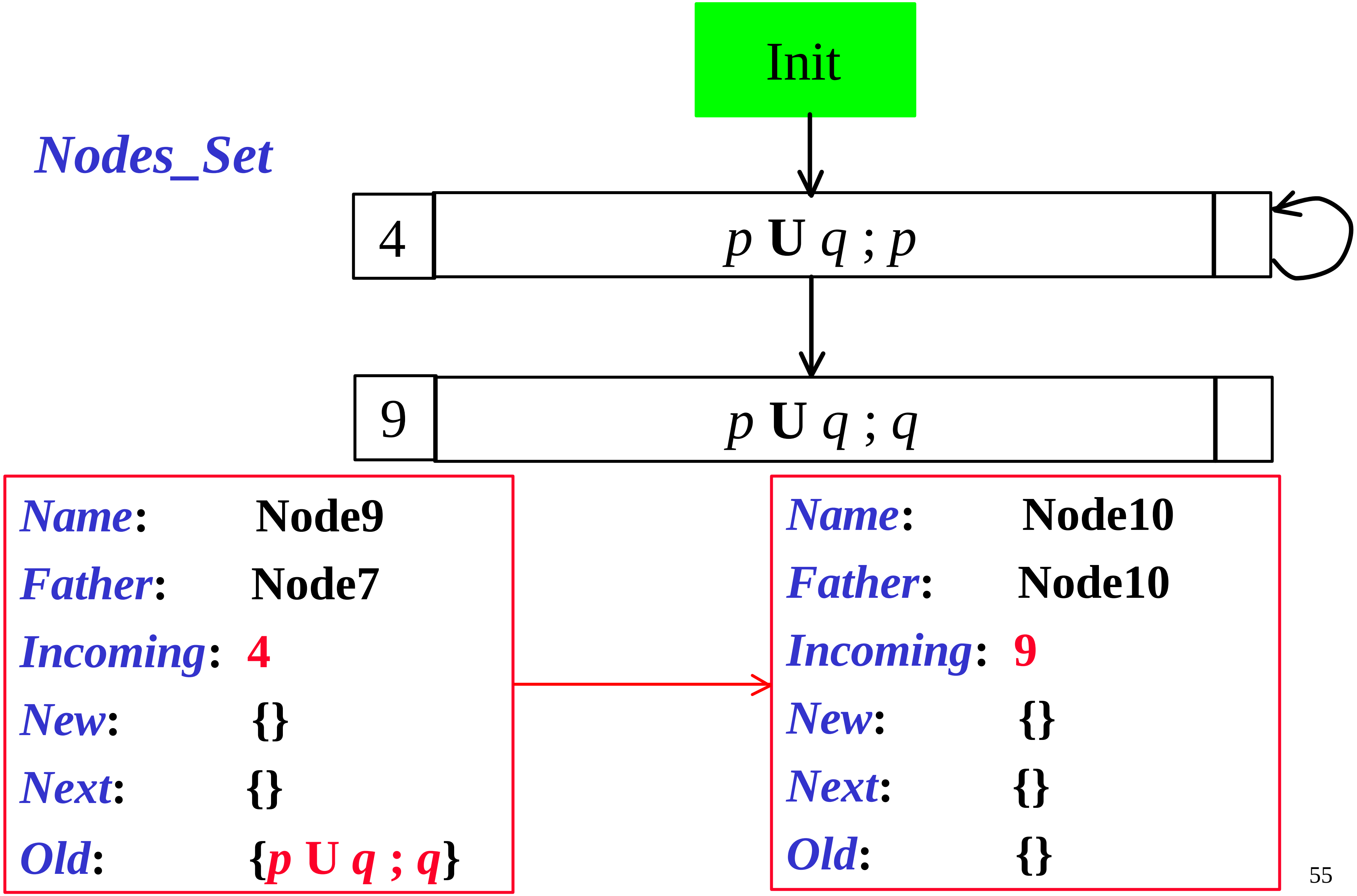
Nodes_Set



Name:	**Node4**
Father:	**Node3**
Incoming:	**Init**
New:	**{}**
Next:	**{*p* U *q*}**
Old:	**{*p* U *q* ; *p*}**

Name:	**Node5**
Father:	**Node5**
Incoming:	**4**
New:	**{*p* U *q*}**
Next:	**{}**
Old:	**{}**

Nodes_Set



Init

4 $p\ \mathbf{U}\ q\ ;\ p$

9 $p\ \mathbf{U}\ q\ ;\ q$

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

Name: **Node10**
Father: **Node10**
Incoming: **9**
New: **{}**
Next: **{}**
Old: **{}**

```
function expand (Node, Nodes_Set)
  if New(Node)=Æ then … /* see previous block */
  else
    let  h ∈ New;
    New(Node) := New(Node) \ {h};
    case h of
      h = p_i or Øp_i or T or ^ :
                    if h = ^ or Neg(h)∈ Old(Node) then
                        return(Nodes_Set) ; /* Discard current node */
                    else Old(Node) ⇐ Old(Node) ∪ {h};
                    return(expand(Node, Nodes Set));
      h = m U y or m R y or m Ú y : ….
```

Contradiction found

Additional functions

The function Neg() is applied only to literals:

Neg(p_i) = Ø p_i **Neg($\top$) = $\wedge$**

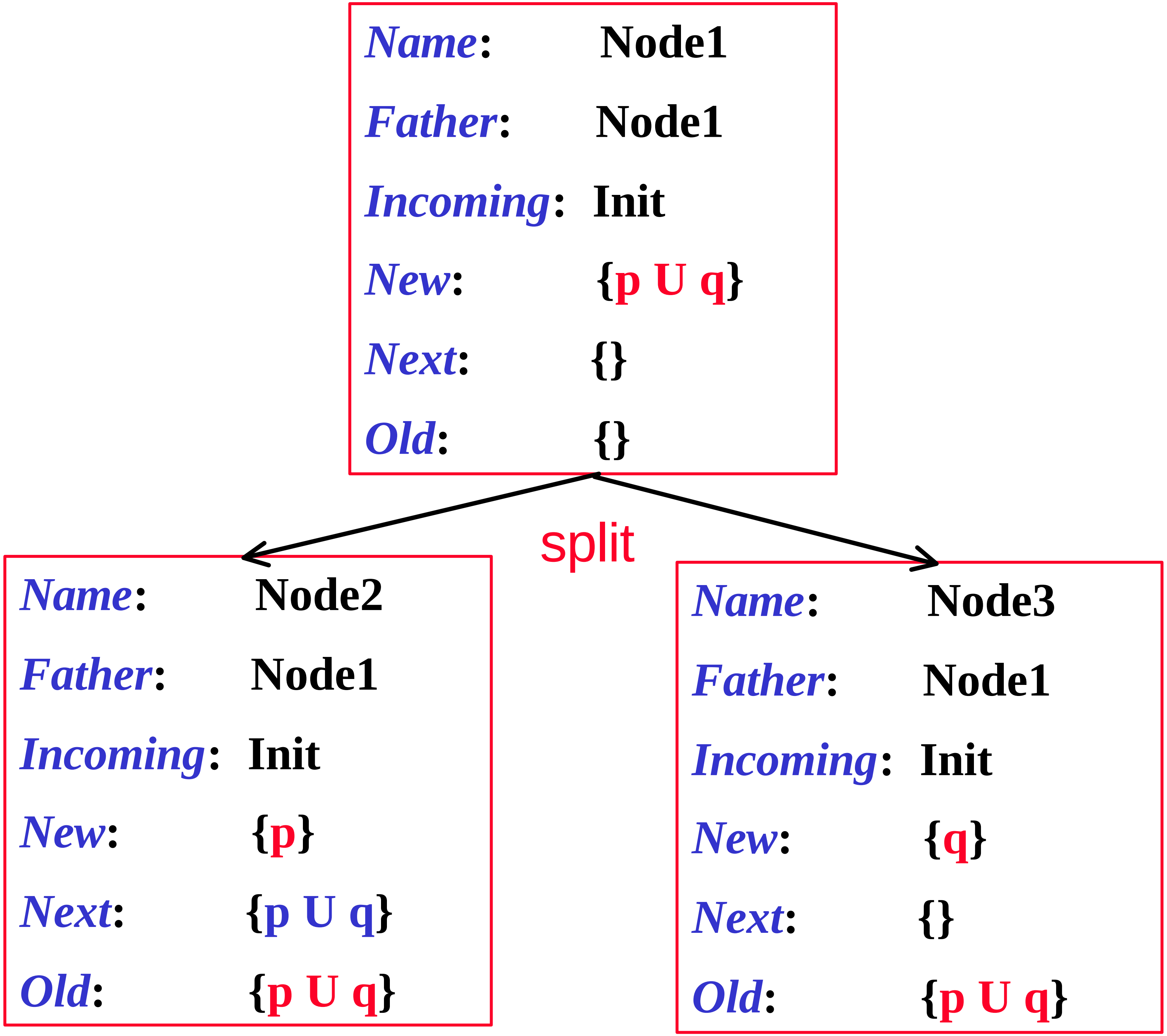
Neg(Øp_i) = p_i **Neg($\wedge$) = $\top$**

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

h	**New1**(h)	**Next1**(h)	**New2**(h)
m **U** y	{m}	{m **U** y }	{ y }
m **R** y	{ y }	{m **R** y }	{m ,y }
m Ú y	{m}	Æ	{ y }

```
function expand (Node, Nodes_Set)
  if New(Node)=Æ then … /* see previous block */
  else
    let  h∈ New;
    New(Node) := New(Node) \ {h};
    case h of
        h = p_i or Øp_i or T or ^: … /* see previous block */
        h = m U y  or m R y   or m Ú y  :
            Node1:=[Name ⇐ new_name(), Father ⇐ Name(Node),
                    Incoming ⇐ Incoming(Node),
                    New ⇐ New(Node) ∪ ({New1(h)} \ Old(Node)),
                    Old ⇐ Old(Node) ∪ {h},
                    Next ⇐ Next(Node) ∪ {Next1(h)}];
            Node2:=[Name ⇐ new_name(), Father ⇐ Name(Node),
                    Incoming ⇐ Incoming(Node),
                    New ⇐ New(Node) ∪ ({New2(h)} \ Old(Node)),
                    Old ⇐ Old(Node) ∪ {h}, Next ⇐ Next(Node)];
            return(expand(Node2, expand(Node1, Nodes_Set)));
        h = m Ù y  : …. /* see next block */
```

splitting



Name: **Node1**

Father: **Node1**

Incoming: **Init**

New: **{p U q}**

Next: **{}**

Old: **{}**

split

Name: **Node2**

Father: **Node1**

Incoming: **Init**

New: **{p}**

Next: **{p U q}**

Old: **{p U q}**

Name: **Node3**

Father: **Node1**

Incoming: **Init**

New: **{q}**

Next: **{}**

Old: **{p U q}**

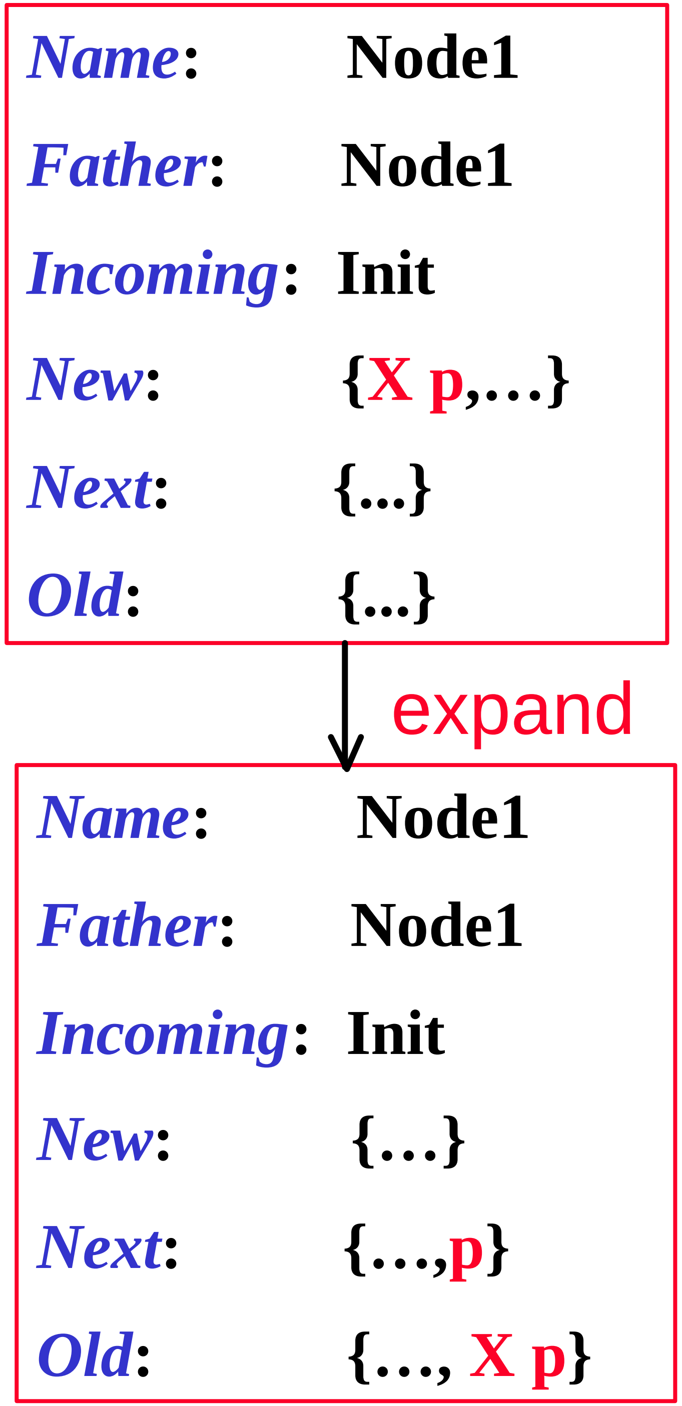
```
function expand (Node, Nodes_Set)
  if New(Node)=Æ then … /* see previous block */
  else
    let h∈ New;
    New(Node):=New(Node) \ {h};
    case h of
        h = $p_i$ or Ø$p_i$ or ⊤ or ^: … /* see previous block */
        h = m U y or m R y or m Ú y : … /* see previous block */
        h = m Ù y :
          return(expand([Name ⇐ Name(Node),
                         Father ⇐ Father(Node),
                         Incoming ⇐ Incoming(Node),
                         New ⇐ (New(Node) ∪ {m ,y } \ Old(Node)),
                        Old ⇐Old(Node) ∪ {h}, Next = Next(Node)],
                 Nodes_Set);
        h = X y : …. /* see next block */
```

***Name*: Node1**
***Father*: Node1**
***Incoming*: Init**
***New*: {p Ù q,…}**
***Next*: {...}**
***Old*: {...}**

expand

***Name*: Node2**
***Father*: Node1**
***Incoming*: Init**
***New*: {p,q,…}**
***Next*: {...}**
***Old*: {…,p Ù q}**

```
function expand (Node, Nodes_Set)
  if New(Node)=Æ then … /* see previous block */
  else
    let h∈ New;
    New(Node):=New(Node) \ {h};
    case h of
          h = $p_i$ or Ø$p_i$ or ⊤ or ^: … /* see previous block */
          h = m U y or m R y or m Ú y : … /* see previous block */
          h = m Ù y : … /* see previous block */
          h = X y :
              return(expand(
                      [Name ⇐ Name(Node),Father ⇐ Father(Node),
                       Incoming ⇐ Incoming(Node), New ⇐ New(Node),
                       Old ⇐ Old(Node) ∪ {h}, Next = Next(Node) ∪ {y}],
                    Nodes_Set);
    esac;
end expand;
```



Name: **Node1**
Father: **Node1**
Incoming: **Init**
New: **{X p,…}**
Next: **{...}**
Old: **{...}**

expand

Name: **Node1**
Father: **Node1**
Incoming: **Init**
New: **{…}**
Next: **{…,p}**
Old: **{…, X p}**

The need for accepting conditions

- *IMPORTANT*: Remember that *not every maximal path* p = $s_0 s_1 s_2$… in the graph *determines a model* of the formula: the construction above allows some node which contain mUy, while none of its successor nodes contain y.
- This is solved again by imposing the *generalized Büchi acceptance conditions* :
  - for each subformula of f of the form m U y , there is a set F_f Î F containing all the nodes *s* Î Q such that either m U y Ï *Old(s)*, or y Î *Old(s)*.

Complexity of the construction

THEOREM: For any LTL formula f a *Büchi automaton* A_f can be constructed which accepts all an only the w-*sequences* (*LTL models*) satisfying f.

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

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

LTL to BA: example

- **Consider the following formula:**

$$\mathbf{G}\, p$$

- **where p is an atomic formula.**
- **Its *negation-normal form* is**

$$\wedge \; \mathbf{R}\, p$$

LTL to BA: example

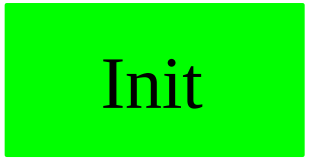
Init

Current node is Node 1
Incoming = [Init]
Old = []
New = [^ **R** *p*]
Next = []

(^ R *p*) º **(*p*** Ù ^**)** Ú
(*p* Ù **X(**^ **R *p*))**

New(node) not empty, removing h = ^ **R** *p*, node ***split*** into 2, 3, about to expand them

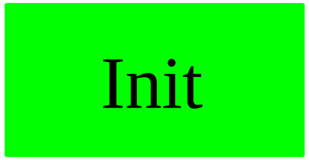
LTL to BA: example



Current node is Node 2
Incoming = [Init]
Old = [$\wedge$ **R** p]
New = [p]
Next = [$\wedge$ **R** p]

New(node) not empty, removing $h = p$, node replaced by 4 about to expand them

LTL to BA: example



Current node is Node 4

Incoming = [Init]

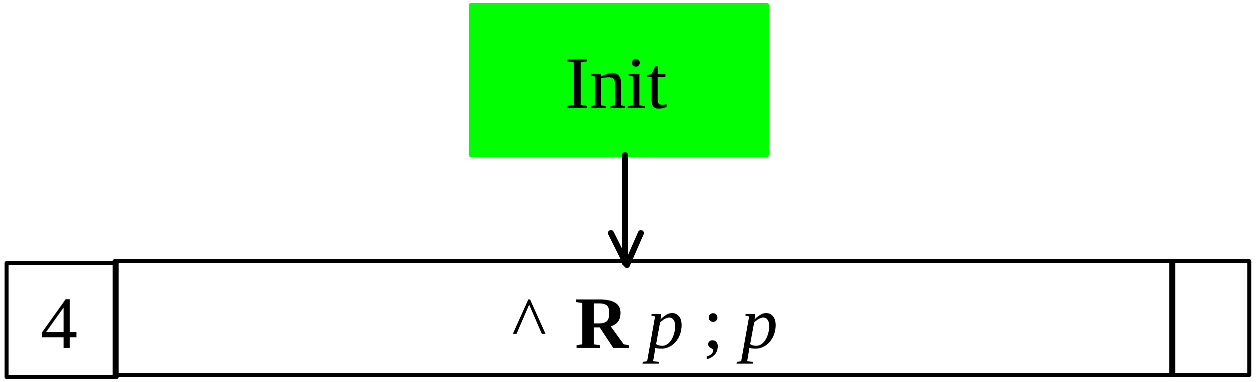
Old = [$\wedge\ \mathbf{R}\ p\ ;\ p$]

New = []

Next = [$\wedge\ \mathbf{R}\ p$]

New(node) empty, no equivalent nodes. About to add, timeshift and expand.

LTL to BA: example

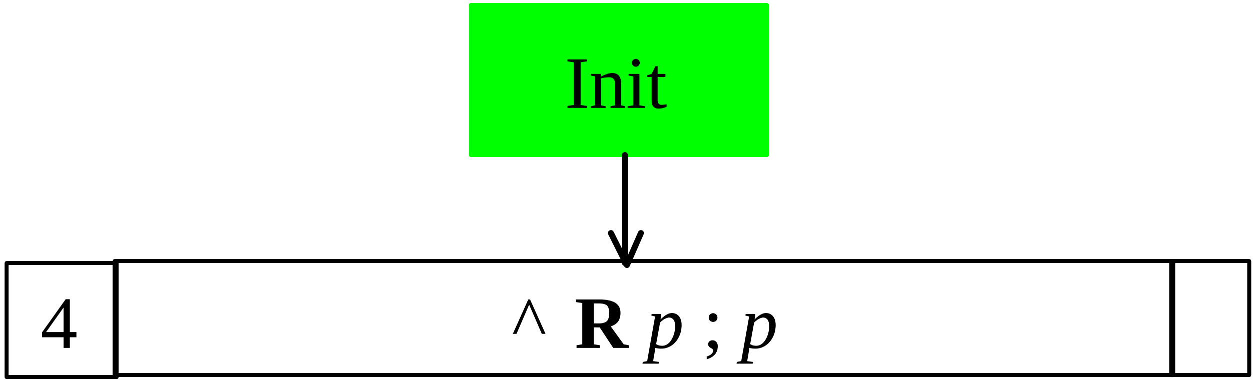


Current node is Node 5
Incoming = [4]
Old = []
New = [^ **R** *p*]
Next = []

(^ **R** ***p***) º (***p*** Ù ^) Ú
(***p*** Ù **X**(^ **R** ***p***))

New(node) not empty, removing h = ^ **R** *p*, node ***split*** into 6, 7 about to expand them

LTL to BA: example



Init

4

$\wedge\ \mathbf{R}\ p\ ;\ p$

Current node is Node 6

Incoming = [4]

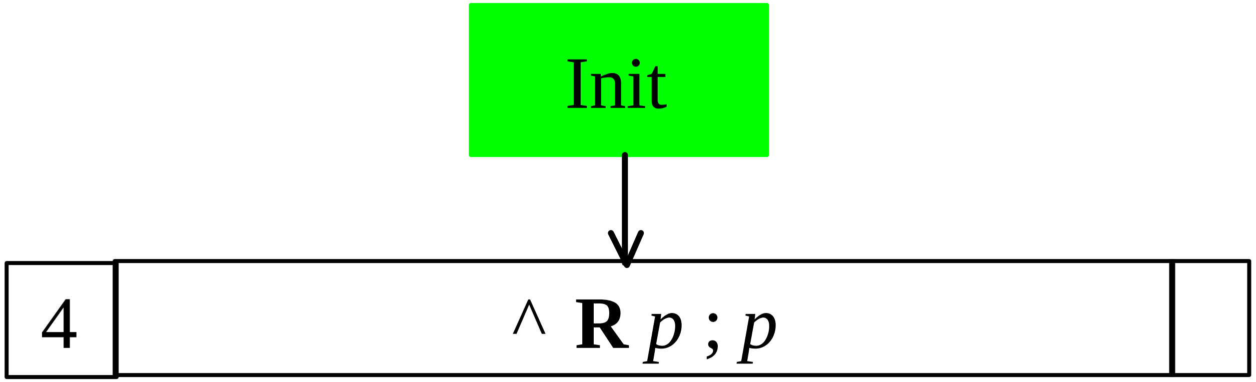
Old = [$\wedge\ \mathbf{R}\ p$]

New = [p]

Next = [$\wedge\ \mathbf{R}\ p$]

New(node) not empty, removing $h = p$, node replaced by 8, about to expand it

LTL to BA: example

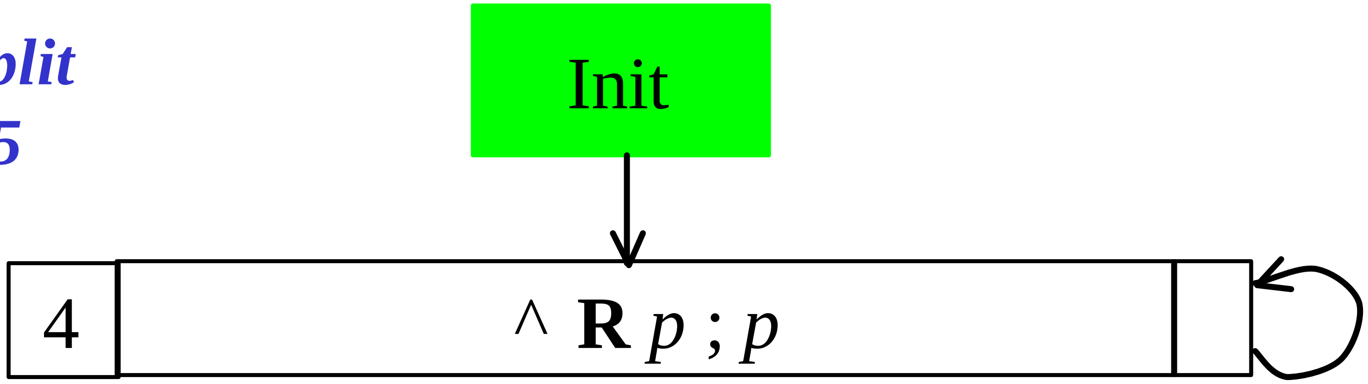


Current node is Node 8
Incoming = [4]
Old = [∧ **R** *p* ; *p*]
New = []
Next = [∧ **R** *p*]

New(node) empty, found equivalent old node in Node_Set (4).
Returning it instead.

LTL to BA: example

From the split of Node 5



Current node is Node 7

Incoming = [4]

Old = [∧ **R** *p*]

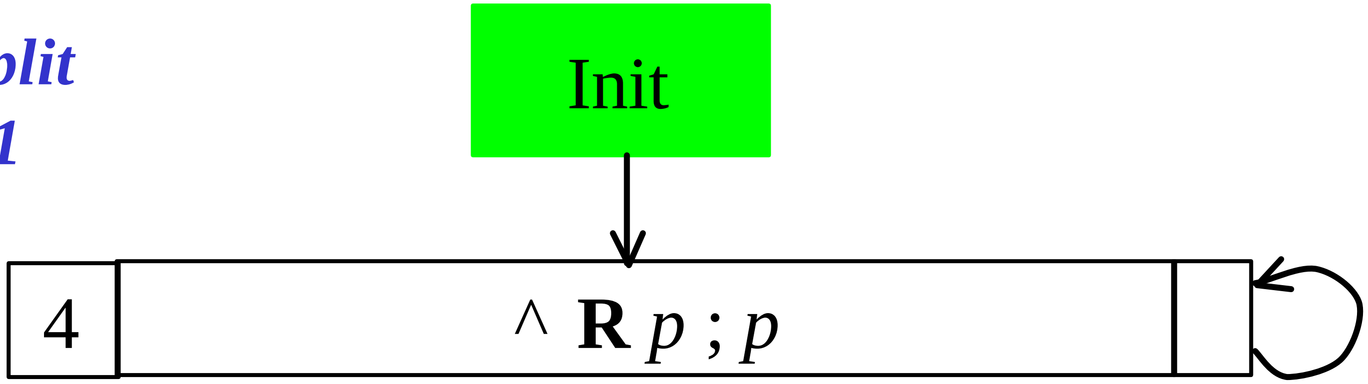
New = [∧ ; *p*]

Next = []

New(node) not empty, removing h = ∧, inconsistent node deleted - dead end!.

LTL to BA: example

From the split of Node 1



Current node is Node 3

Incoming = [Init]

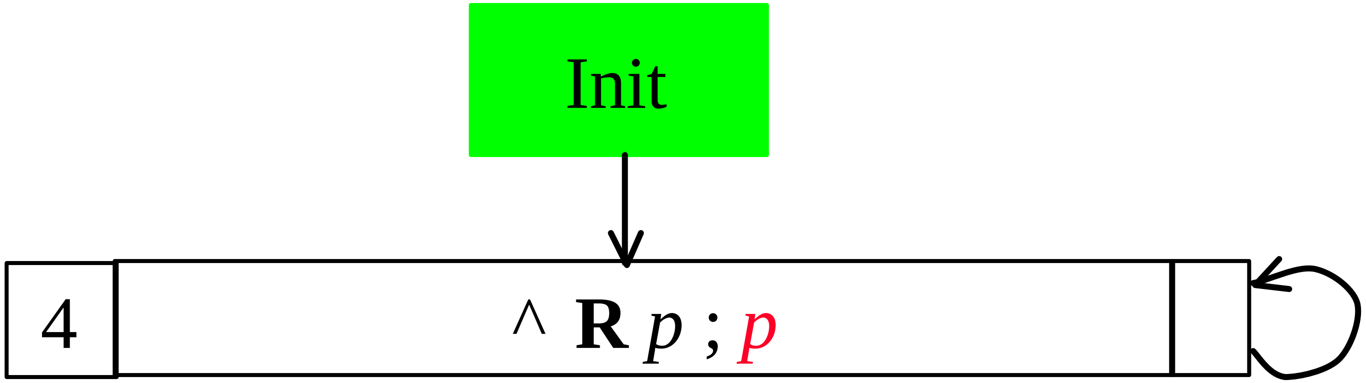
Old = [^ **R** *p*]

New = [^ ; *p*]

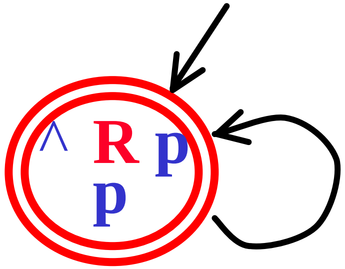
Next = []

New(node) not empty, removing h = ^, inconsistent node deleted - dead end!.

LTL to BA: example



Final graph for ***G p*** o ^ ***R p***



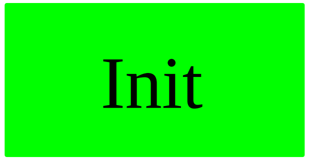
LTL to BA: example 2

Consider the following formula:

$$p \mathbf{U} q$$

where p and q are atomic formulae.

LTL to BA: example 2



Current node is Node 1
Incoming = [Init]
Old = []
New = [*p* U *q*]
Next = []

(*p* U *q*) ° ***q*** Ú **(*p*** Ù **X(*p* U *q*))**

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

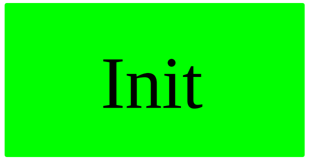
LTL to BA: example 2

Init

Current node is Node 2
Incoming = [Init]
Old = [$p \ \mathrm{U} \ q$]
New = [p]
Next = [$p \ \mathrm{U} \ q$]

New(node) not empty, removing $h = \boldsymbol{p}$ node replaced by 4, about to expand them

LTL to BA: example 2



Init

Current node is Node 4

Incoming = [Init]

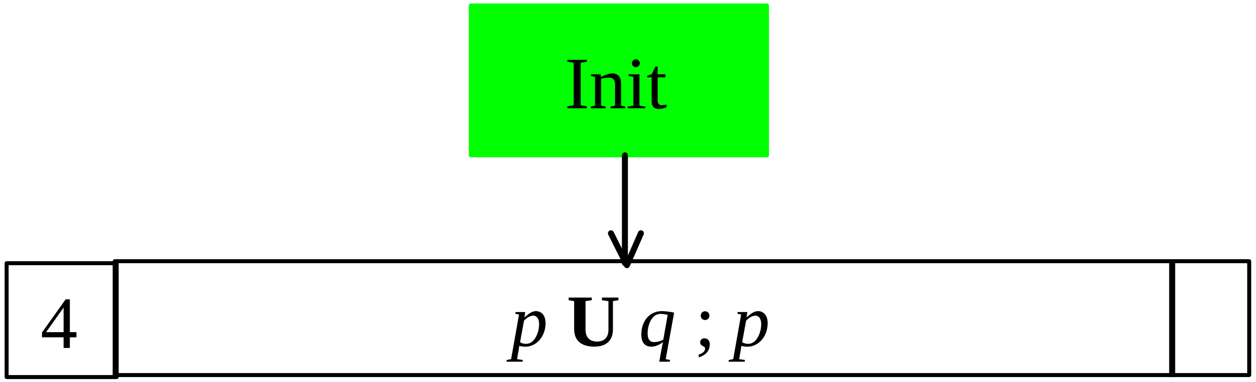
Old = [p U q ; p]

New = []

Next = [p U q]

New(node) empty, no equivalent nodes. Add, timeshift and expand.

LTL to BA: example 2



Current node is Node 5

Incoming = [4]

Old = []

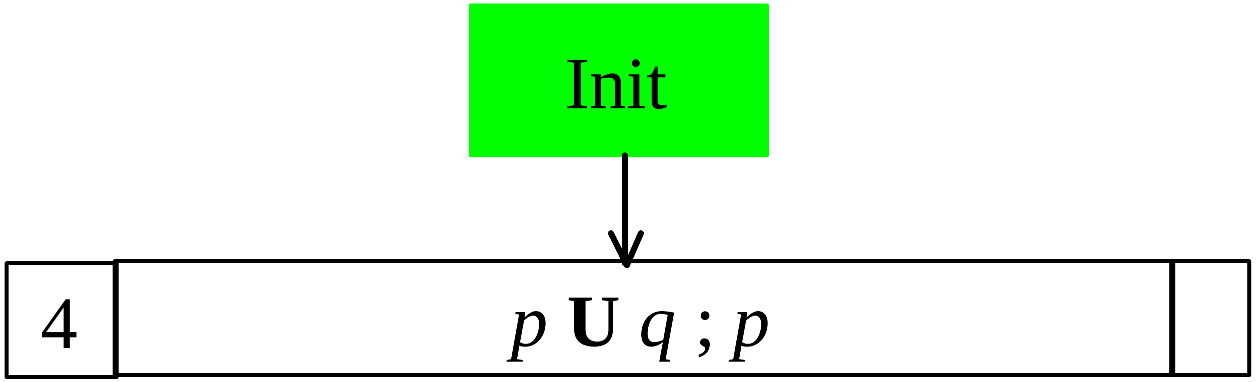
New = [$p \mathrm{U} q$]

Next = []

$(\boldsymbol{p} \mathbf{U} \boldsymbol{q}) \equiv \boldsymbol{q} \vee (\boldsymbol{p} \wedge \mathbf{X}(\boldsymbol{p} \mathbf{U} \boldsymbol{q}))$

New(node) not empty, removing $\eta = \boldsymbol{p} \mathbf{U} \boldsymbol{q}$, node ***split*** into 6 , 7, about to expand.

LTL to BA: example 2



Init

4

$p \; \mathbf{U} \; q \; ; \; p$

Current node is Node 6

Incoming = [4]

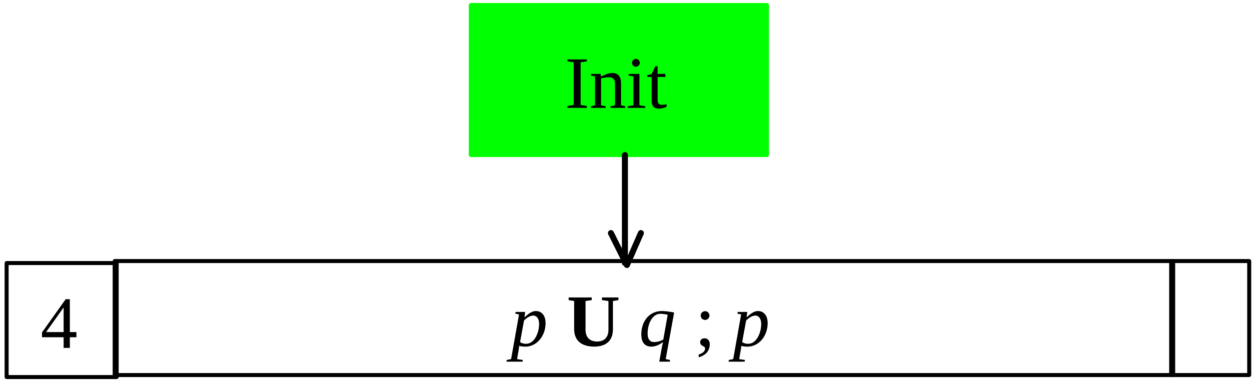
Old = [p U q]

New = [p]

Next = [p U q]

New(node) not empty, removing h = $\boldsymbol{p}$, node replaced by 8, about to expand it

LTL to BA: example 2



Current node is Node 8

Incoming = [4]

Old = [p U q ; p]

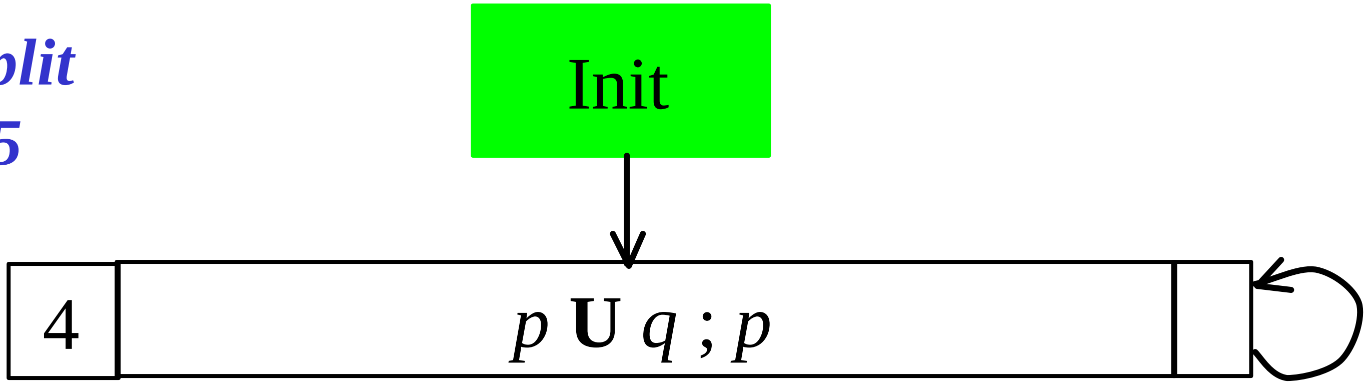
New = []

Next = [p U 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



Init

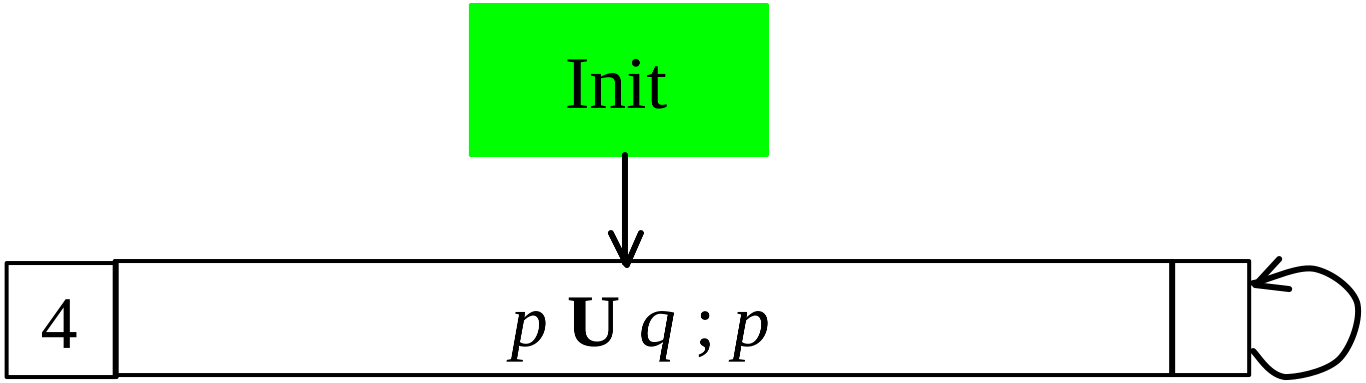
4

$p \; \mathbf{U} \; q \; ; \; p$

Current node is Node 7
Incoming = [4]
Old = [p U q]
New = [q]
Next = []

New(node) not empty, removing h = q, node replaced by 9, about to expand it

LTL to BA: example 2



Current node is Node 9

Incoming = [4]

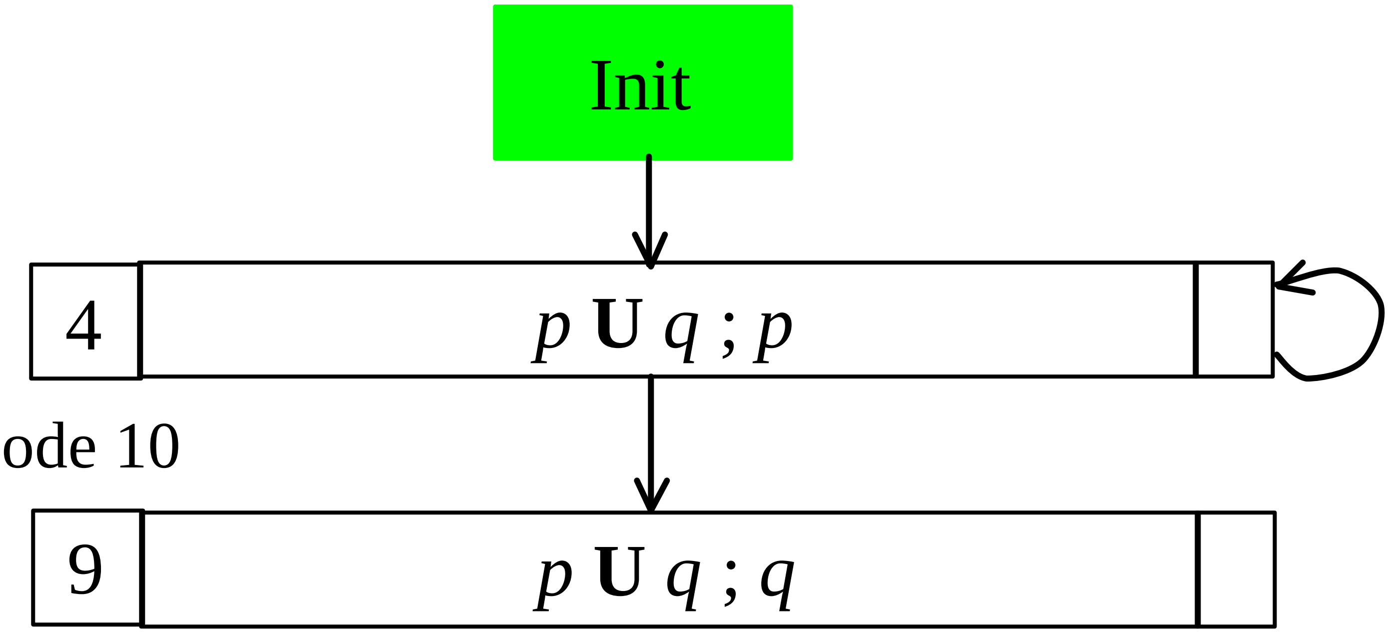
Old = [p U q ; q]

New = []

Next = []

New(node) empty, no equivalent node found. Add timeshift and expand

LTL to BA: example 2



Init

4

$p \mathbf{U} q ; p$

9

$p \mathbf{U} q ; q$

Current node is Node 10

Incoming = [9]

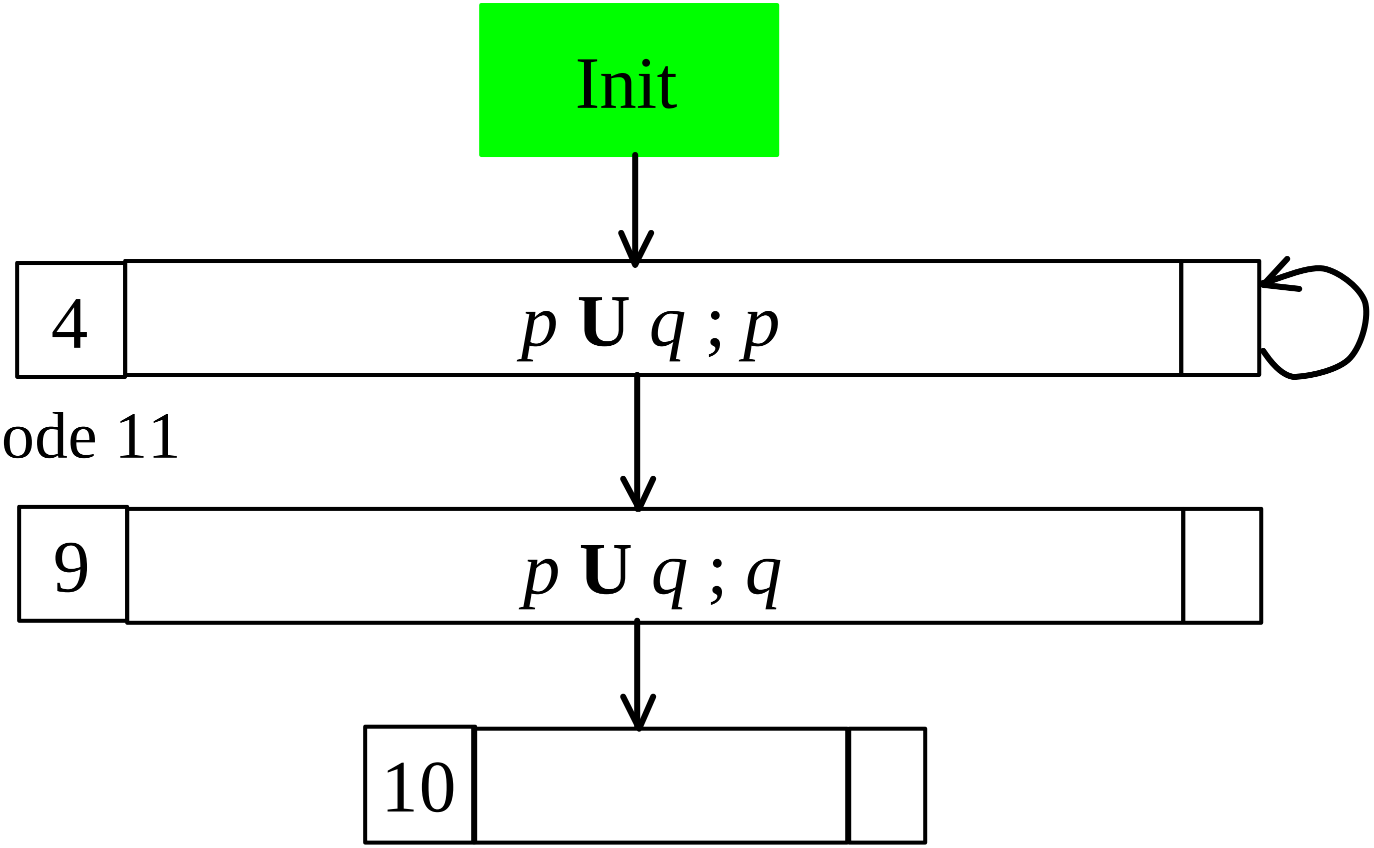
Old = []

New = []

Next = []

New(node) empty, no equivalent node found. Add timeshift and expand

LTL to BA: example 2

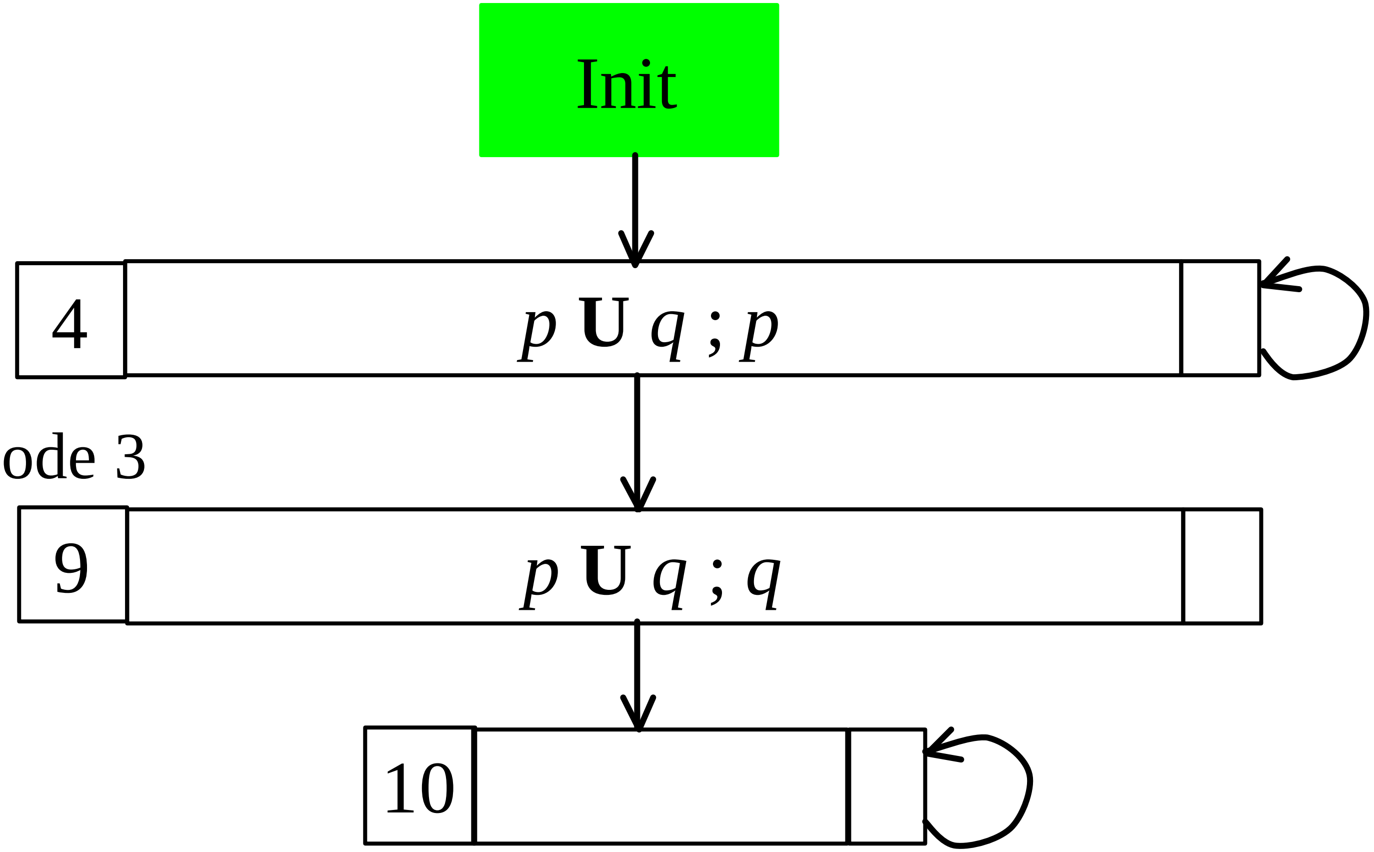


Current node is Node 11
Incoming = [10]
Old = []
New = []
Next = []

New(node) empty. Found equivalent old node in Node_Set (10). Returning it instead.

LTL to BA: example 2

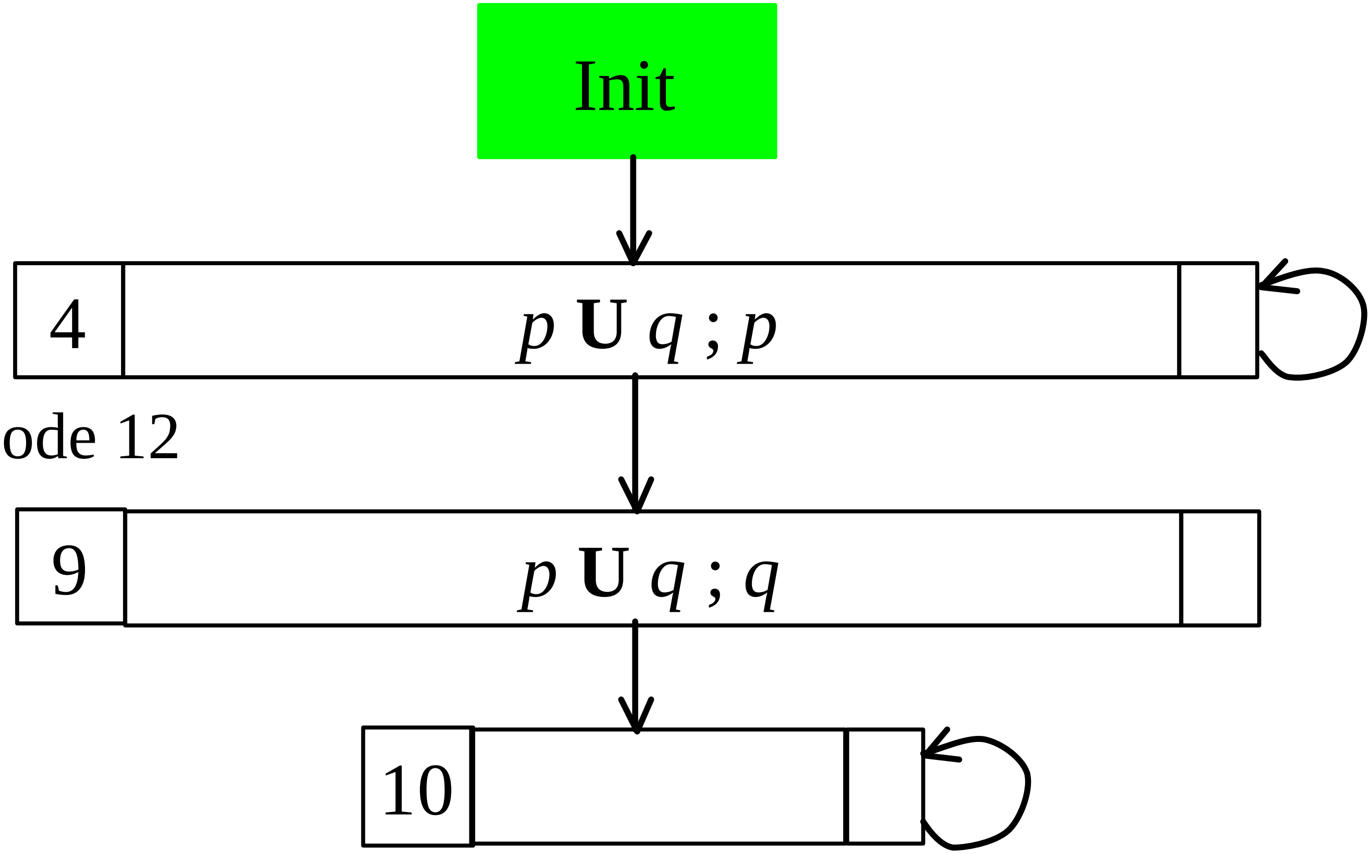
From the split of Node 1



Current node is Node 3
Incoming = [Init]
Old = [p **U** 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]

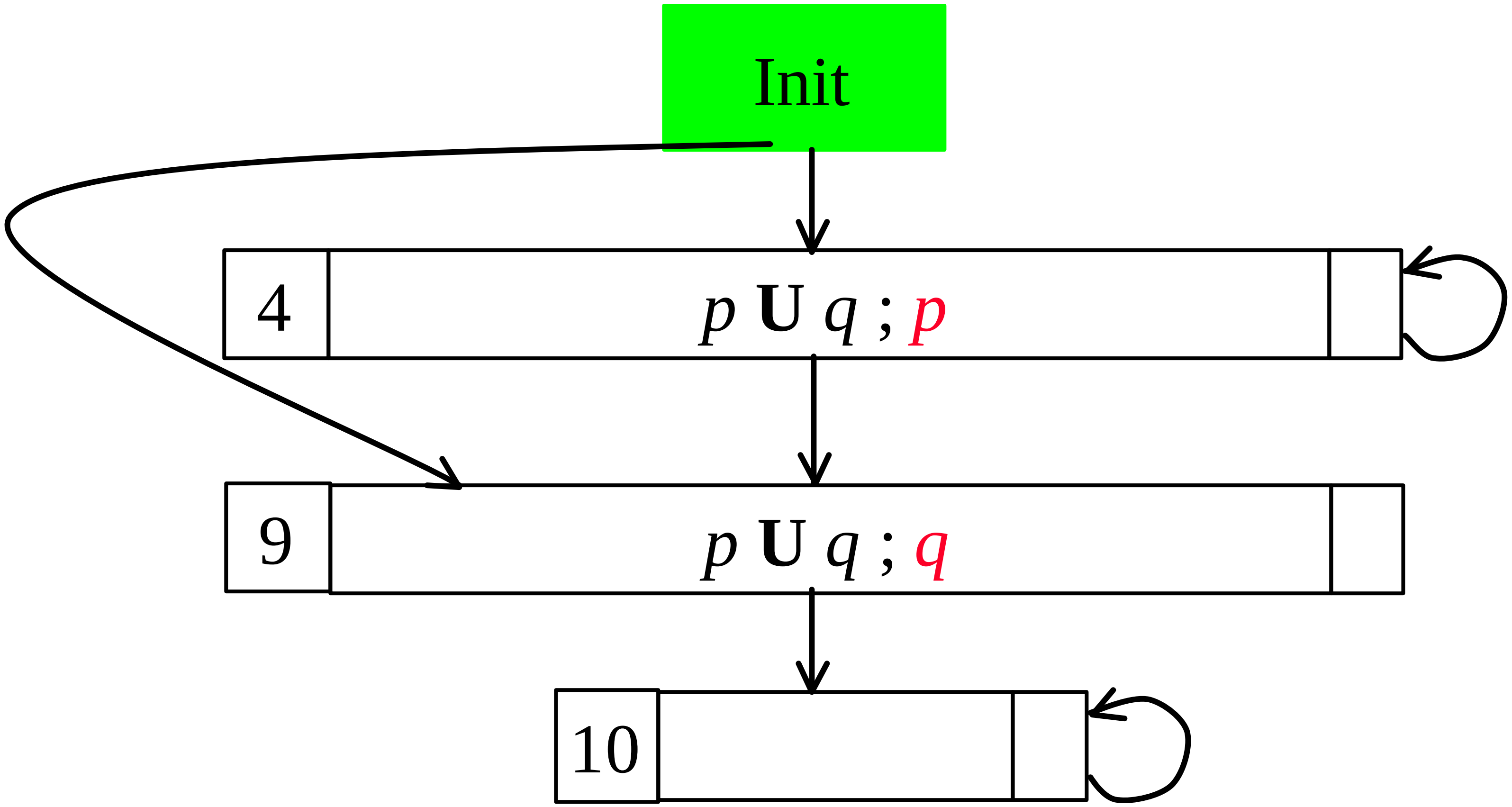
Old = [p **U** q ; q]

New = []

Next = []

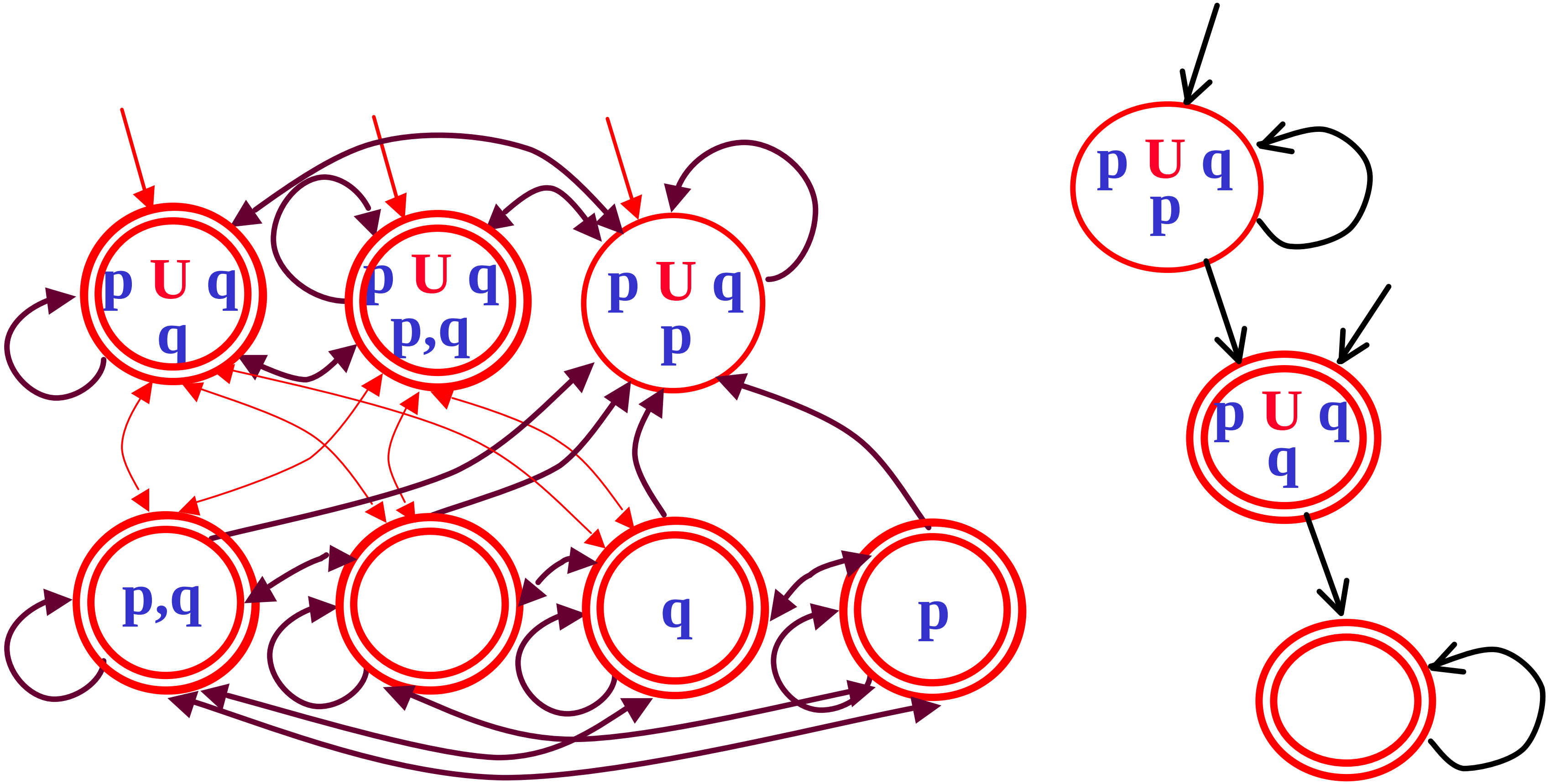
New(node) empty. Found equivalent old node (4) in Node_Set. Returning it instead.

LTL to BA: example 2



Final graph for ***p* U *q***

Comparison of the two algorithms



The graphs for ***p* U *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)!**

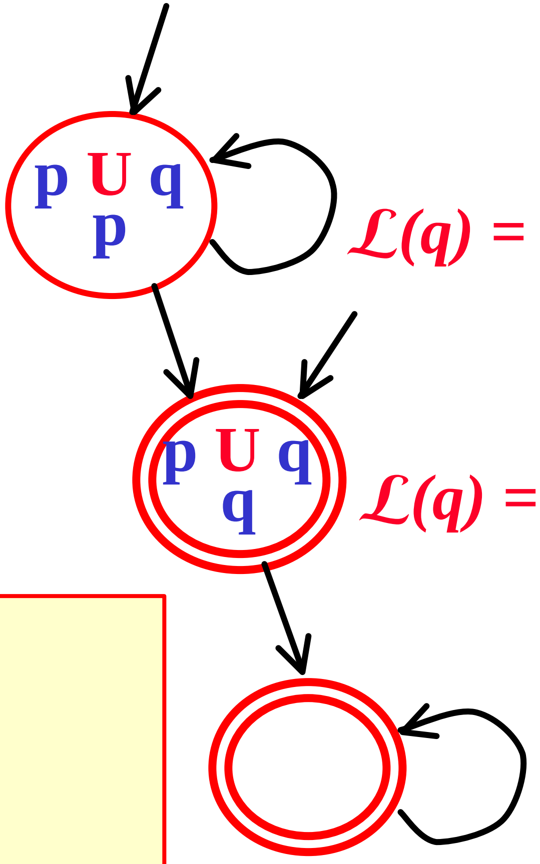
- **Indeed the *labeling* to be associated to a node can be *any element of* 2^{AP} which agrees with the *literals* (AP or negations of AP) in *Old(Node)*.**

- **Let *Pos(q)* = *Old(q)* Ç AP**

- **Let *Neg(q)* = {h Î AP| Øh Î *Old(q)*}**

L*(q)* = { X Í AP | X Ê *Pos(q)* Ù (X Ç *Neg(q)*) = Æ}

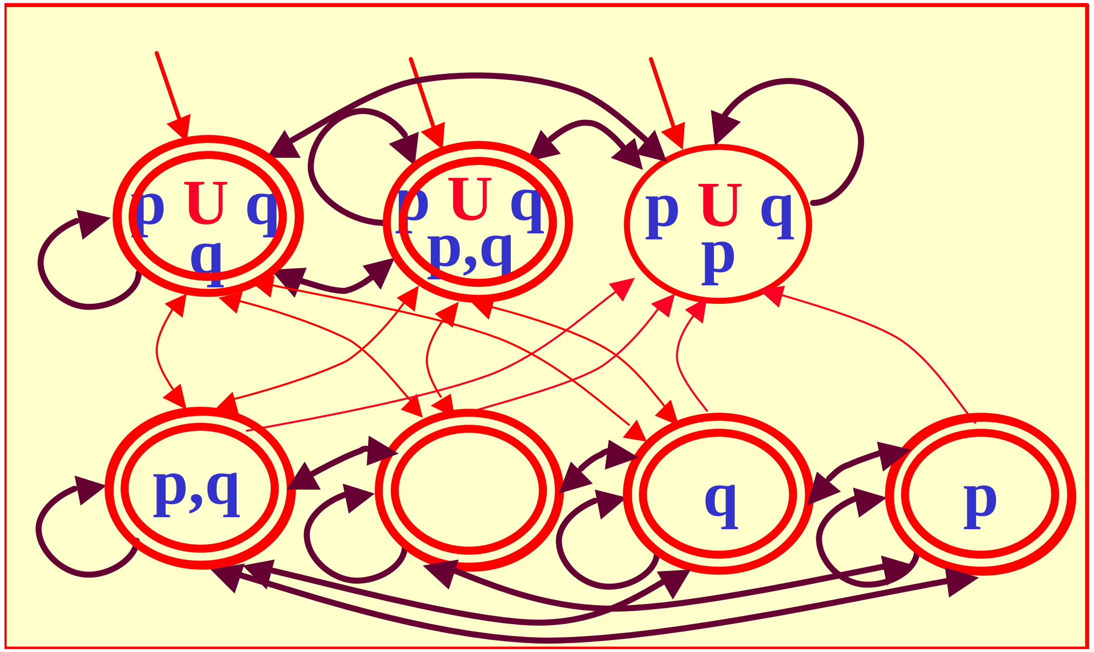
Notes on the algorithm



$\mathcal{L}(q) = \{\{p\},\{p,q\}\}$

$\mathcal{L}(q) = \{\{q\},\{p,q\}\}$

$\mathcal{L}(q) = \{\{\},\{p\},\{q\},\{p,q\}\}$



Composing A_{sys} and A_f

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

Efficient composition of A_{sys} and A_f

- **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} Ç A_f = < S, S_{sys} ´ S_f, R', S_{0sys} ´ S_{0f}, S_{sys} ´ F_f >
- **where**

  (<*s*,*t*>,*a*,<*s'*,*t'*>) Î R' *iff* (*s*,*a*,*s'*) Î R_{sys} and (*t*,*a*,*t'*) Î R_f

Efficient composition of A_{sys} and A_f

- **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**

$$A_{sys} \cap A_f = < S, S', R', (S_{0sys} \times S_{0f}) \cap S', (S_{sys} \times F_f) \cap S' >$$

- **where**

$$S' = \{(s,t) \in S_{sys} \times S_f \mid L_{sys}(s)|_{AP_f} = L_f(t)\} \text{ and}$$

$$(<s,t>,<s',t'>) \in R' \quad \textit{iff} \quad (s,s') \in R_{sys} \quad \text{and} \quad (t,t') \in R_f$$