SPOT

What is SPOT?

Spot **P**roduces **O**ur **T**races

"An extensible object-oriented model-checking library written in C++."

1. INTRODUCTION
2. AUTOMATA
3. MODEL CHECKING
4. EXAMPLES

INTRODUCTION

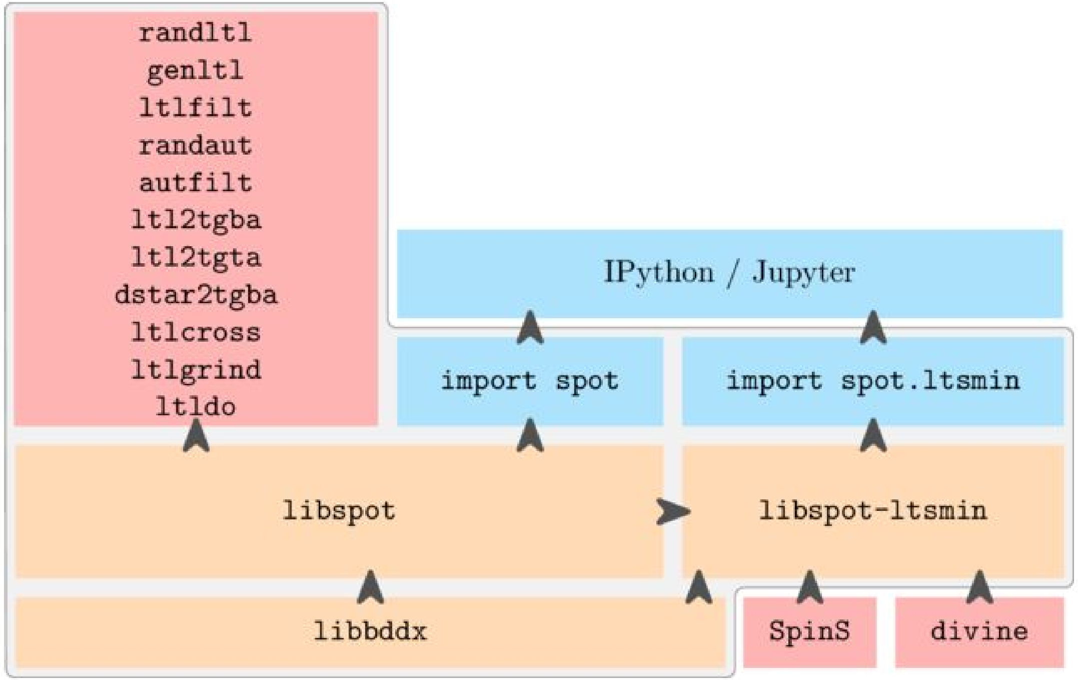
Part 1/4

What does SPOT provide?

Several languages :

- Python (and Ipython/Jupyter)
- C++
- Command lines

And one **online tool** for formula translation



What does SPOT provide?

Supports 4 syntax for automata's description:

- Never Claims (by Spin)
- LBTT (by LBT)
- DSTAR
- HOA (Hanoi Omega-Automaton)

```
HOA: v1
name: "FGp0 | GFp1"
States: 4
Start: 0
AP: 2 "p0" "p1
acc-name: Rabin 2
Acceptance: 4 (Fin(0) & Inf(1)) | (Fin(2) &
Inf(3))
properties: trans-labels explicit-labels state-
acc complete
properties: deterministic
--BODY--
State: 0 {0}
[!0&!1] 0
[0&!1] 1
[!0&1] 2
[0&1] 3
State: 1 {1}
[!0&!1] 0
[0&!1] 1
[!0&1] 2
[0&1] 3
--END--
```

```
never { /* p0 | GFp1 */
T0_init:
  do
  :: atomic { (p0) -> assert(!(p0))
  :: (!(p0)) -> goto accept_S2
  od;
accept_S2:
  do
  :: (p1) -> goto accept_S2
  :: (!(p1)) -> goto T0_S3
  od;
T0_S3:
  do
  :: (p1) -> goto accept_S2
  :: (!(p1)) -> goto T0_S3
  od;
accept_all:
  skip
}
```

```
DRA v2 explicit
Comment: "Union{Safra[NBA=2],Safra[NBA=2]}"
States: 4
Acceptance-Pairs: 2
Start: 0
AP: 2 "p0" "p1"
---
State: 0
Acc-Sig: -0
0
1
2
3
State: 1
Acc-Sig: +0
0
1
2
```

```
3 1t
0 1
1 -1 p0
2 -1 ! p0
-1
1 0
1 0 -1 t
-1
2 0
2 0 -1 p1
2 -1 ! p1
-1
```

What does SPOT provide?

2 LTL syntax :

- LTL
- PSL (Property Specification Language)

LTL formula	meaning
`f`	the formula **f** is true immediately
`X f`	**f** will be true in the next step
`F f`	**f** will become true eventually (it could be true immediately, or on the future)
`G f`	**f** is always true from now on
`f U g`	**f** has to be true until **g** becomes true (and **g** *will* become true)
`f W g`	**f** has to be true until **g** becomes true (**f** should stay true if **g** never becomes true)
`f R g`	**g** has to be true until **f&g** becomes true (**g** should stay true if **f&g** never becomes true)
`f M g`	**g** has to be true until **f&g** becomes true (and **f&g** *will* become true)

PSL formula	meaning
`{e}<>->f`	**f** should hold on the last instant of some one prefix that matches **e**
`{e}[]->f`	**f** should hold on the last instant of all prefixes that match **e**

Advantages

- No modus operandi, the library has no hard-wired operating procedure
  → *“bricks” to build a model checker*
  → *extensive*

- SPOT relies on automata called Transition-based Generalized Büchi Automata (TGBA)
  → *they allow more compact translations of LTL*

- Operation are implemented by several algorithms

Usages

- Transformation LT formula → Automata
- LT formula manipulation : simplifying, testing equivalence,...
- Automata manipulation : emptiness check, product,...
- Automata transformation

AUTOMATA

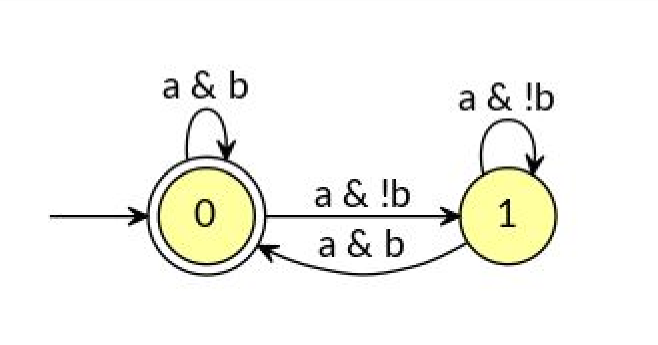
Part 2/4

Different kind of Büchi Automata

1- Büchi Automaton:

- w-automaton
- accept if it visits some accepting state infinitely often

Example:



→Accept only if :

- a is always true
- b is infinitely often true

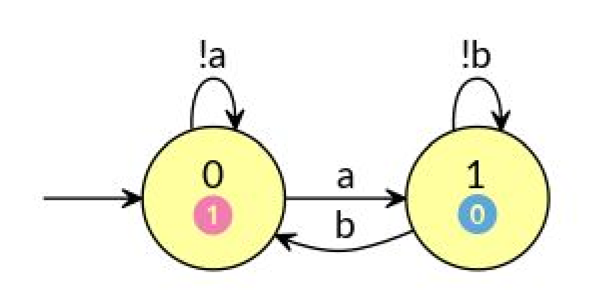
Different kind of Büchi Automata

2- Generalized-Büchi Automaton:

- multiple sets of accepting sets called acceptance sets
- generalized Büchi acceptance condition:

  → a run is accepting iff it visits at least one state of each acceptance set.

Example:



→Accept only if :

- a is infinitely often true
- b is infinitely often true

Transition-based and State-based acceptance

1- State-based acceptance:

As we have seen so far :

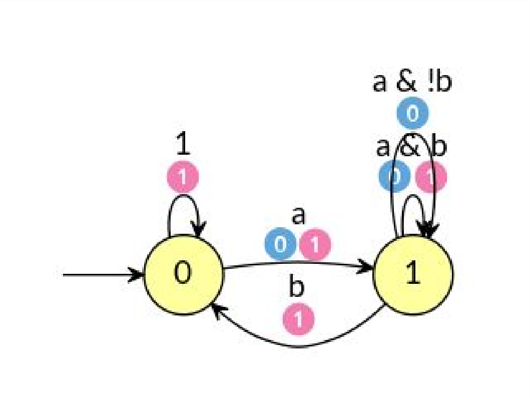
-accepting if it visit infinitely often some state in each acceptance set

2- Transition-based acceptance:

-Acceptance set : set of edges (or transitions)

-Accept if :

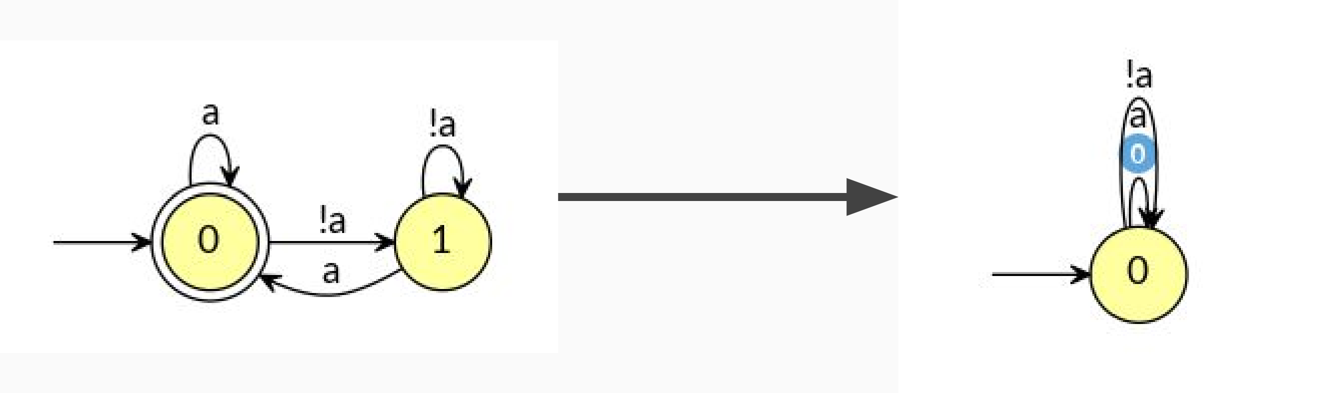
-accepting if it visit infinitely often some transition in each acceptance set



→Accept all ω-words that infinitely often match the pattern a+;b

Transition-based and State-based acceptance

3- Translation from State-based automata to Transition-based automata:



-2 states with use of transition-based automata
-1 state with use of state-based automata

Acceptance condition

Possibility to work with general forms of acceptance condition

Acceptance condition = 2 pieces :

- -Acceptance sets
- -Formula that tell how to use the acceptance sets.

Acceptance formulas are positive Boolean formula over atoms of the form **t**, **f, Inf(n)**, or **Fin(n)**, where n is a non-negative integer denoting an acceptance set.

- **t** denotes the true acceptance condition: any run is accepting
- **f** denotes the false acceptance condition: no run is accepting
- **Inf(n)** means that a run is accepting if it visits infinitely often the acceptance set **n**
- **Fin(n)** means that a run is accepting if it visits finitely often the acceptance set **n**

Combination with & and |

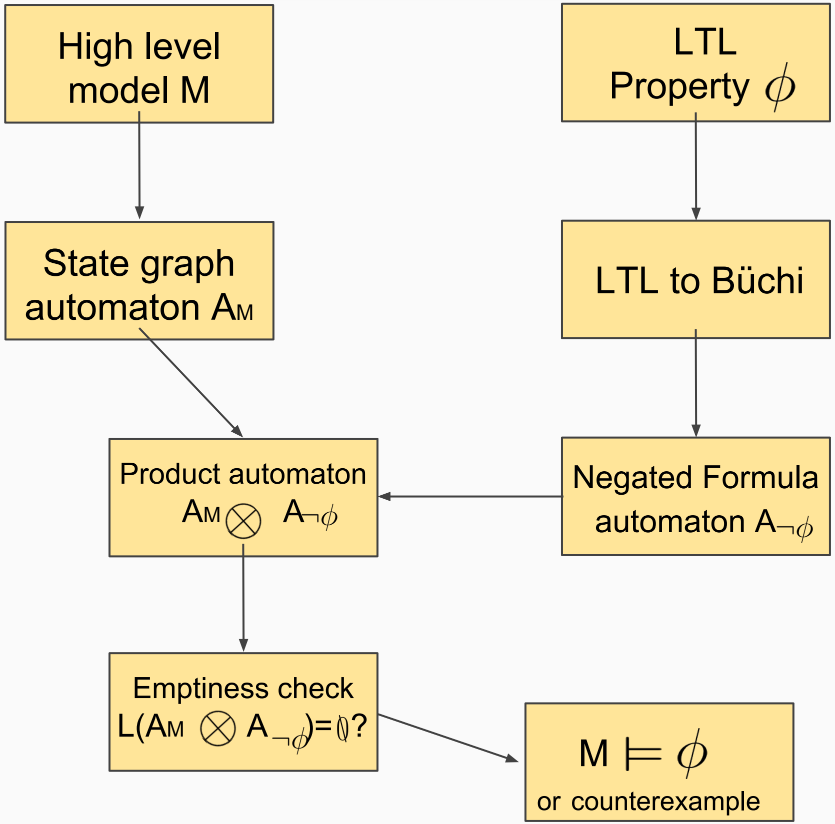
<table>
<tr><td>none</td><td>f</td></tr>
<tr><td>all</td><td>t</td></tr>
<tr><td>Buchi</td><td>Inf(0)</td></tr>
<tr><td>generalized-Buchi 2</td><td>Inf(0)&Inf(1)</td></tr>
<tr><td>generalized-Buchi 3</td><td>Inf(0)&Inf(1)&Inf(2)</td></tr>
<tr><td>co-Buchi</td><td>Fin(0)</td></tr>
<tr><td>generalized-co-Buchi 2</td><td>Fin(0)|Fin(1)</td></tr>
<tr><td>generalized-co-Buchi 3</td><td>Fin(0)|Fin(1)|Fin(2)</td></tr>
<tr><td>Rabin 1</td><td>Fin(0) & Inf(1)</td></tr>
<tr><td>Rabin 2</td><td>(Fin(0) & Inf(1))|(Fin(2) & Inf(3))</td></tr>
<tr><td>Rabin 3</td><td>(Fin(0) & Inf(1))|(Fin(2) & Inf(3))|(Fin(4) & Inf(5))</td></tr>
<tr><td>Streett 1</td><td>Fin(0)|Inf(1)</td></tr>
<tr><td>Streett 2</td><td>(Fin(0)|Inf(1)) & (Fin(2)|Inf(3))</td></tr>
<tr><td>Streett 3</td><td>(Fin(0)|Inf(1)) & (Fin(2)|Inf(3)) & (Fin(4)|Inf(5))</td></tr>
<tr><td>generalized-Rabin 3 1 0 2</td><td>(Fin(0) & Inf(1))|Fin(2)|(Fin(3) & (Inf(4)&Inf(5)))</td></tr>
<tr><td>parity min odd 5</td><td>Fin(0) & (Inf(1)|(Fin(2) & (Inf(3)|Fin(4))))</td></tr>
<tr><td>parity max even 5</td><td>Inf(4)|(Fin(3) & (Inf(2)|(Fin(1) & Inf(0))))</td></tr>
</table>

Table presenting some classical acceptance conditions

MODEL CHECKING

Part 3/4

Model Checkers and the Automata-Theoretic Approach (Reminder)



Four principal operations :

- Computation of the state graph of the model M

  $L(A_M)$ all possible executions of the system

- Translation of the temporal property ϕ into a ω-automaton + negation

  $L(A_{\neg\phi})$ all executions that would invalidate ϕ

- Product $A_M \otimes A_{\neg\phi}$

  $L(A_M) \cap L(A_{\neg\phi})$ the set of executions of the model M that invalidate the temporal property ϕ

- Emptiness check of the product.

  This operation tells whether $A_M \otimes A_{\neg\phi}$ accepts an infinite word, and can return such a word (a counter-example). The model M verifies ϕ iff $L(A_M \otimes A_{\neg\phi}) = \varnothing$

On-the-fly model checking

- The computation of the product, state graph, and formula automaton are all driven by the progression of the emptiness-check procedure: nothing is computed until it is required

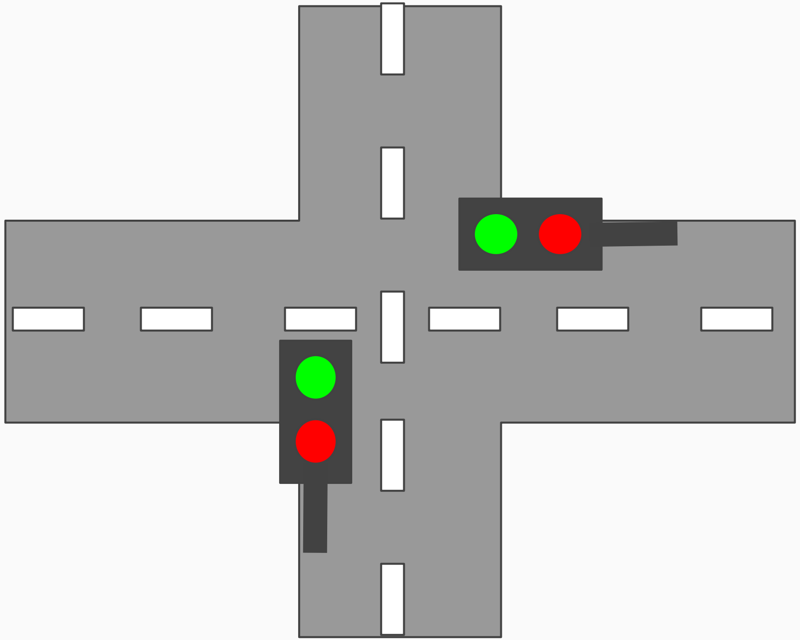
- One of Spot’s design goal is to implement each step of this automata-theoretic approach independently, so they can be combined or replaced at will by users.

  → That does not preclude on-the-fly computations

EXAMPLE

Part 4/4

Example



Example

High level model M

n =

```
%%dve n


process Feu1 {
  state vert, rouge;
  init vert;
  trans
    vert -> rouge {},
    rouge -> vert {guard Feu2.vert == 0;};
}

process Feu2 {
  state vert, rouge;
  init rouge;
  trans
    vert -> rouge {},
    rouge -> vert {guard Feu1.vert == 0;};
}

system async;
```

We define here two traffic lights.
They switch between green and red.
They never turn green when the other one already is.

Example

High level model M

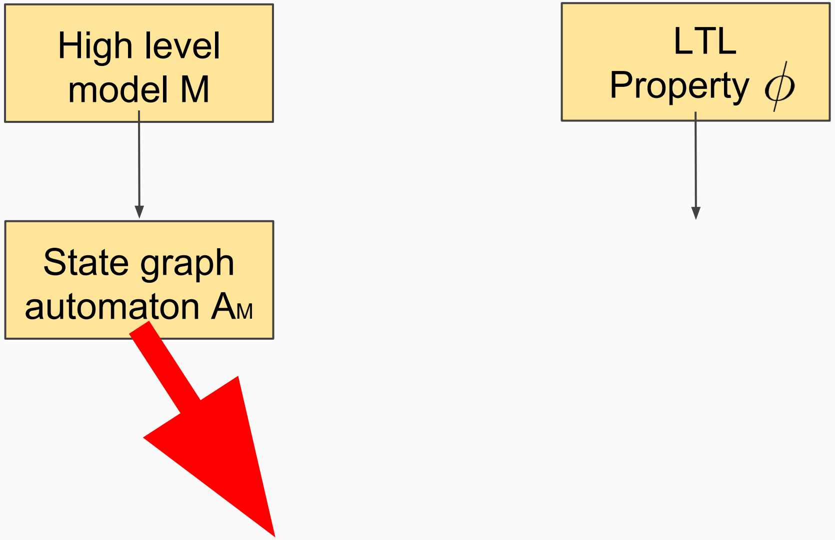
LTL Property ϕ

f =

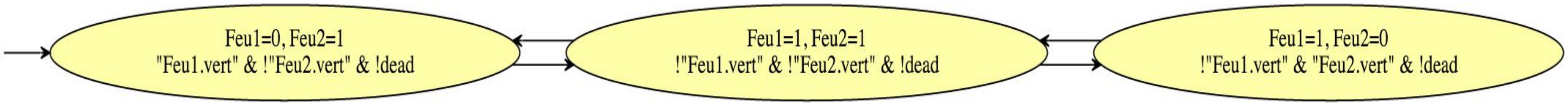
[]!("Feu1.vert"&&"Feu2.vert")

This LTL means that there will never be the two traffic lights green at the same time.

Example



ss = n.kripke(spot.atomic_prop_collect(f))



Example

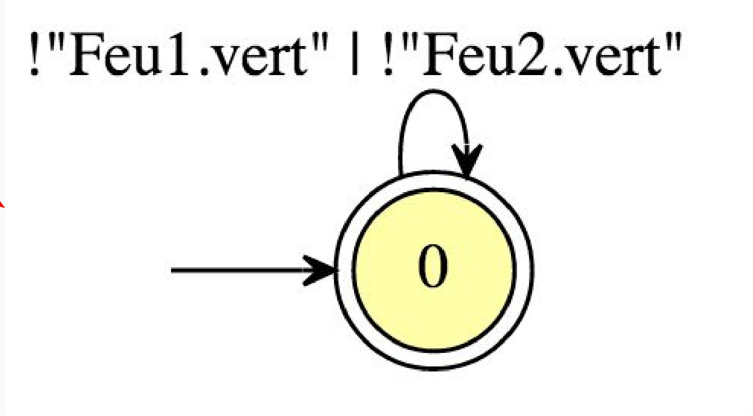
High level model M

State graph automaton A_M

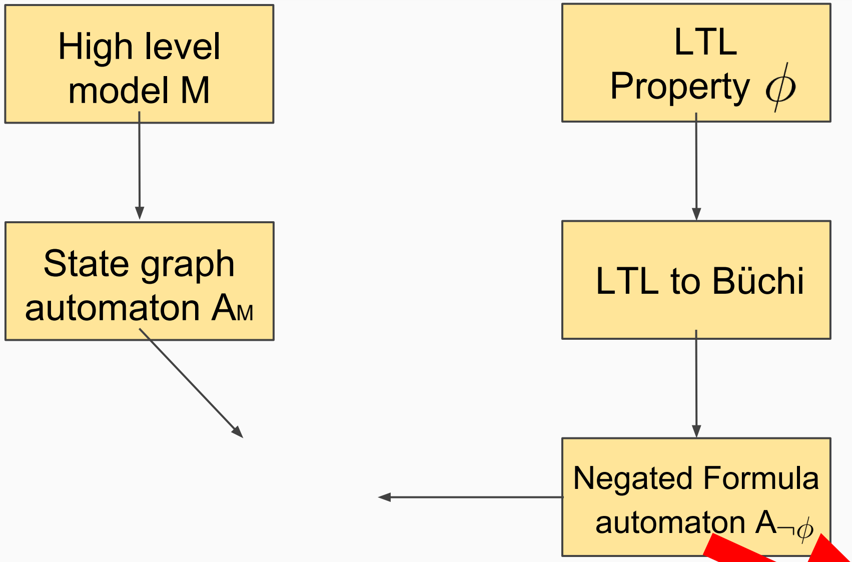
LTL Property ϕ

LTL to Büchi

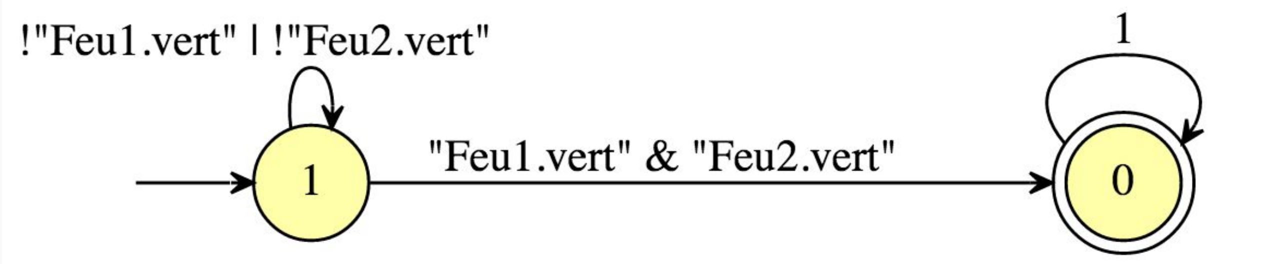
f’ = spot.translate(f)



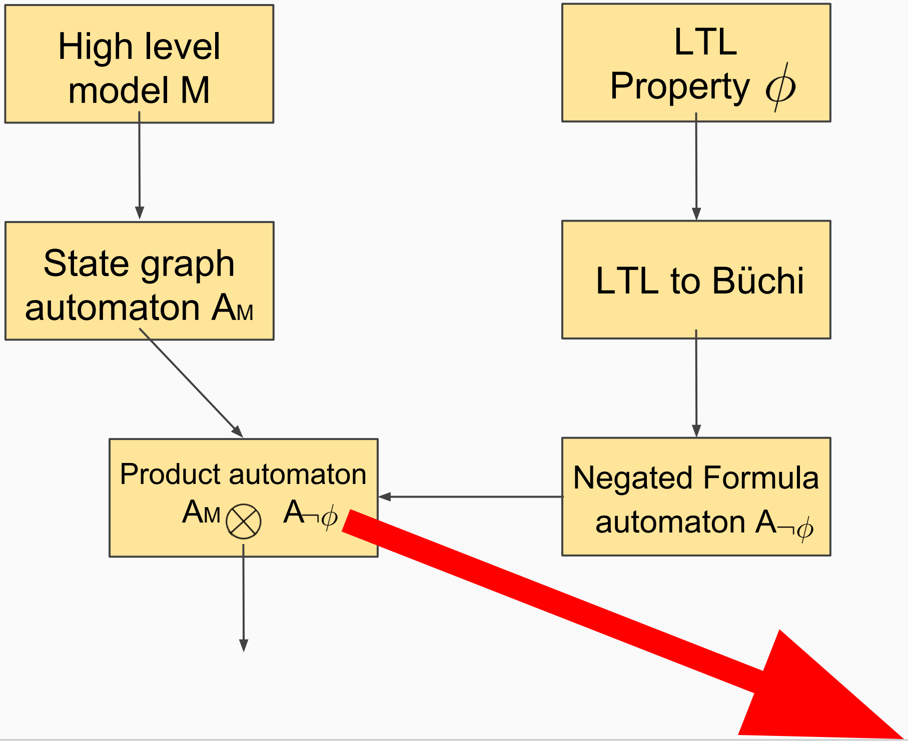
Example



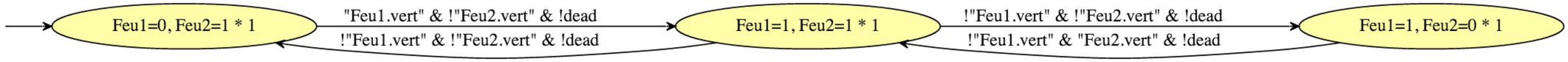
nf = spot.formula_Not(f’).translate()



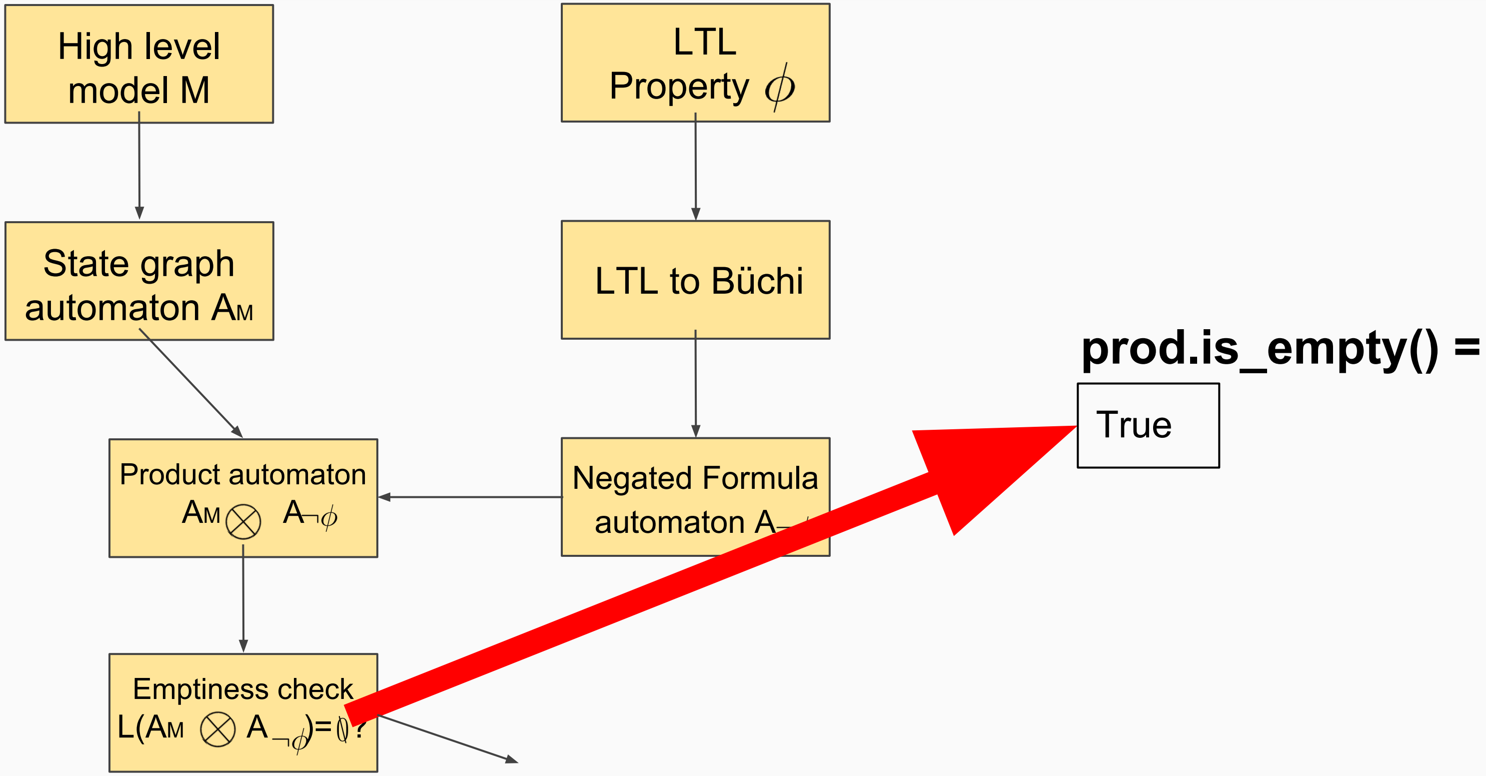
Example



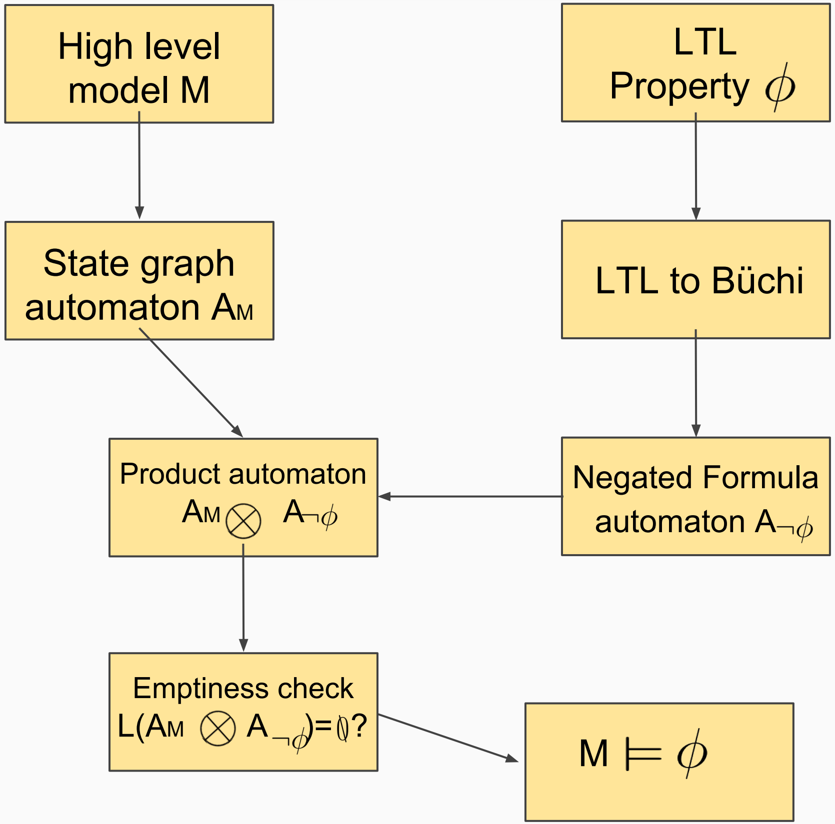
prod = spot.otf_product(ss, nf)



Example


High level model M
LTL Property ϕ
State graph automaton A_M
LTL to Büchi
prod.is_empty() =
True
Product automaton $A_M \otimes A_{\neg\phi}$
Negated Formula automaton $A_{\neg\phi}$
Emptiness check $L(A_M \otimes A_{\neg\phi})=\emptyset$?

Example



M verifies the LTL property ϕ

Example

What if the LTL formula isn't verified ?

```
# state-acceptance Buchi:
f2 = spot.formula('[]!("Feu1.rouge"&&"Feu2.rouge")')
nf2 = spot.formula_Not(f2).translate()
ss2 = n.kripke(spot.atomic_prop_collect(f2))
prod2 = spot.otf_product(ss2, nf2)
prod2
```

```
false
```

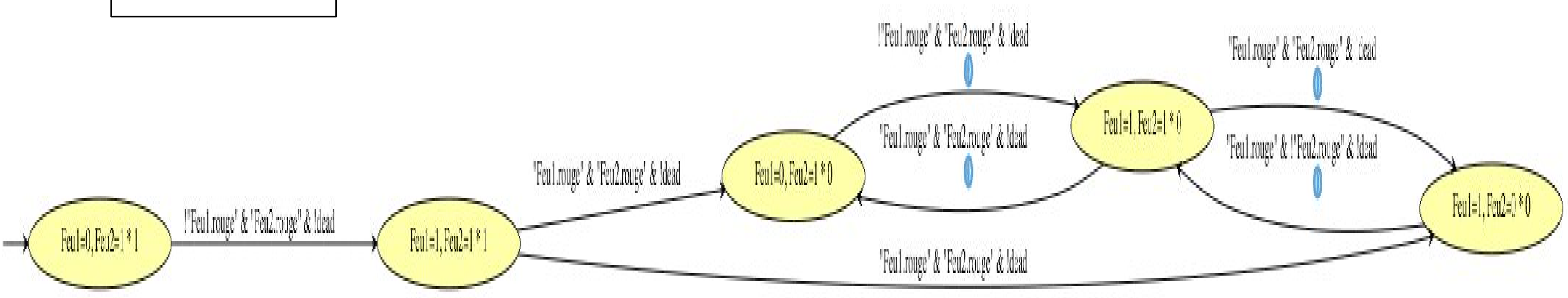
We ask here if the two traffic light will never be red at the same time

The answer is no. They can be red at the same time

Example

We can find why:

```
prod2.show()
```



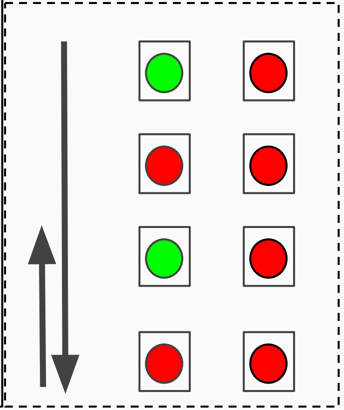
This Tωa contains accepting transitions

Example

We can find a counter-example:

```
run = spot.couvreur99(prod2).check().accepting_run(); run
```

```
Prefix:
  Feu1=0, Feu2=1 * 1
  |  !"Feu1.rouge" & "Feu2.rouge" & !dead
  Feu1=1, Feu2=1 * 1
  |  "Feu1.rouge" & "Feu2.rouge" & !dead
Cycle:
  Feu1=0, Feu2=1 * 0
  |  !"Feu1.rouge" & "Feu2.rouge" & !dead        {0}
  Feu1=1, Feu2=1 * 0
  |  "Feu1.rouge" & "Feu2.rouge" & !dead         {0}
```



THANK YOU !