Spin LTL model checking

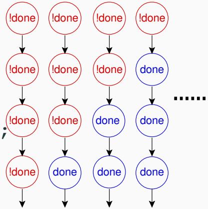
Enrico Magnago

University of Trento,
Fondazione Bruno Kessler

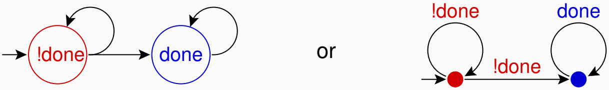
LTL model checking: introduction

- the behaviour of a system $\mathcal{M}$ is given by the set of all its possible paths of execution
  $\bigcup \pi_i = s_{i,0} \rightarrow s_{i,1} \rightarrow \ldots \rightarrow s_{i,t} \rightarrow \ldots$

```
bool done = false;
do
    :: done;
    :: else ->
        if
            :: true -> done = true;
            :: true -> skip;
        fi
od;
```



- The set of computations can be represented by a finite automaton



LTL model checking: Spin

GOAL: verify whether $\mathcal{M} \models \phi$

1. Build Automatons:
   - $A_{\mathcal{M}}$: encodes all possible executions of $\mathcal{M}$
   - $A_{\neg\phi}$: encodes all violations of ϕ
   - $A_{\mathcal{M}\times\neg\phi} = A_{\mathcal{M}} \times A_{\neg\phi}$: contains all the paths in $\mathcal{M}$ that violate ϕ

   ($\times$: synchronous product)

2. Check for a possible execution π_i of $A_{\mathcal{M}\times\neg\phi}$:
   - if π_i exists, then it is a violation *(counter-example)* of ϕ in $\mathcal{M}$.
   - otherwise, $\mathcal{M} \models \phi$.

LTL model checking: Spin

GOAL: verify whether $\mathcal{M} \models \phi$

1. Build Automatons:
   - $A_{\mathcal{M}}$: encodes all possible executions of $\mathcal{M}$
   - $A_{\neg\phi}$: encodes all violations of ϕ
   - $A_{\mathcal{M}\times\neg\phi} = A_{\mathcal{M}} \times A_{\neg\phi}$: contains all the paths in $\mathcal{M}$ that violate ϕ

     ($\times$: synchronous product)

2. Check for a possible execution π_i of $A_{\mathcal{M}\times\neg\phi}$:
   - if π_i exists, then it is a violation *(counter-example)* of ϕ in $\mathcal{M}$.
   - otherwise, $\mathcal{M} \models \phi$.

Important: $\mathcal{M} \models \phi$ iff $\forall i.\pi_i \models \phi$

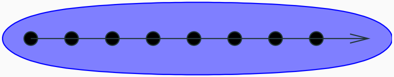
$\Longrightarrow$ not sufficient to check whether there exists a π_i for $A_{\mathcal{M}\times\phi}$

LTL Basics

finally p

F p

globally p

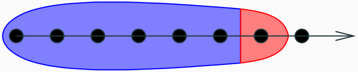


G p

next p

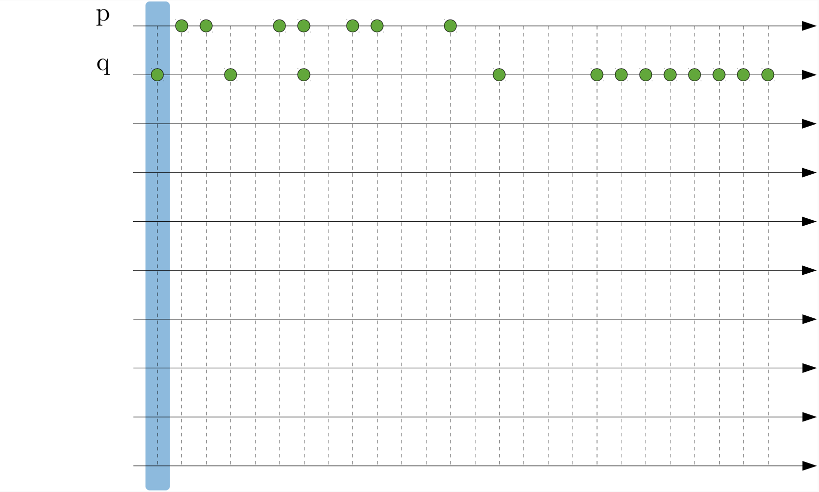
X p

p until q

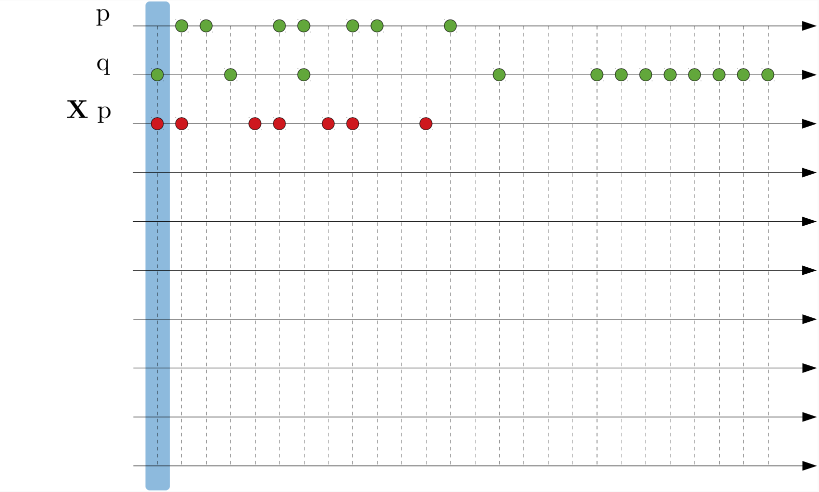


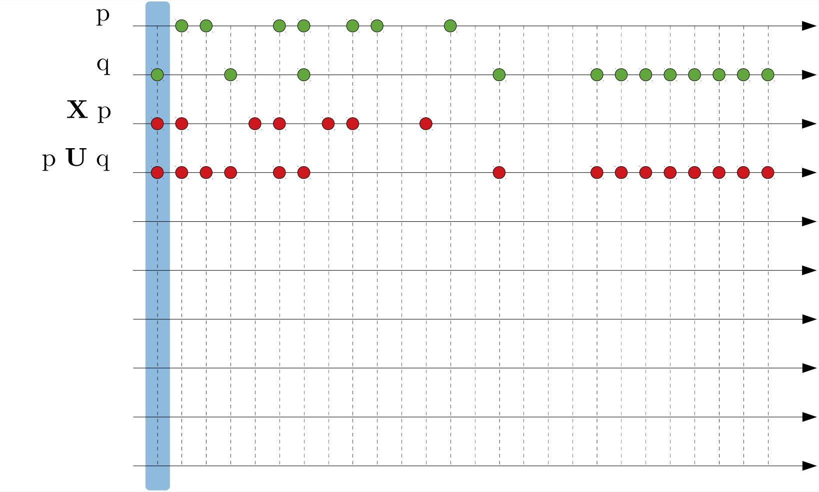
p U q

Execution Model & LTL Properties [1/9]

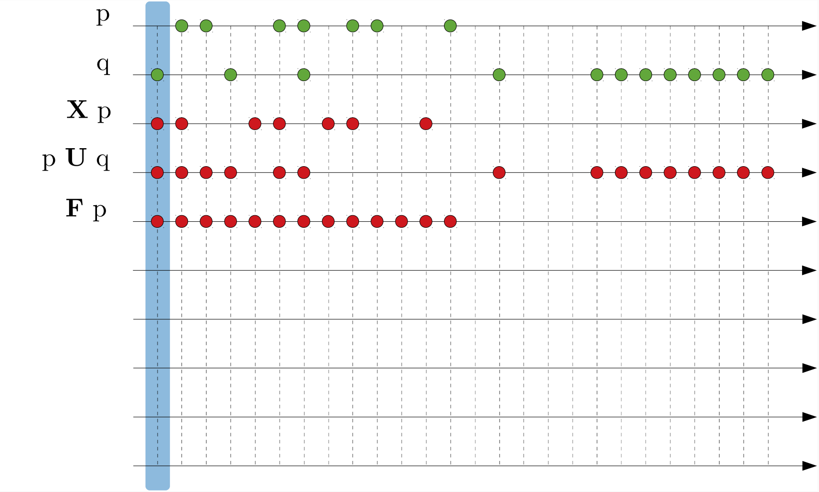


Execution Model & LTL Properties [2/9]

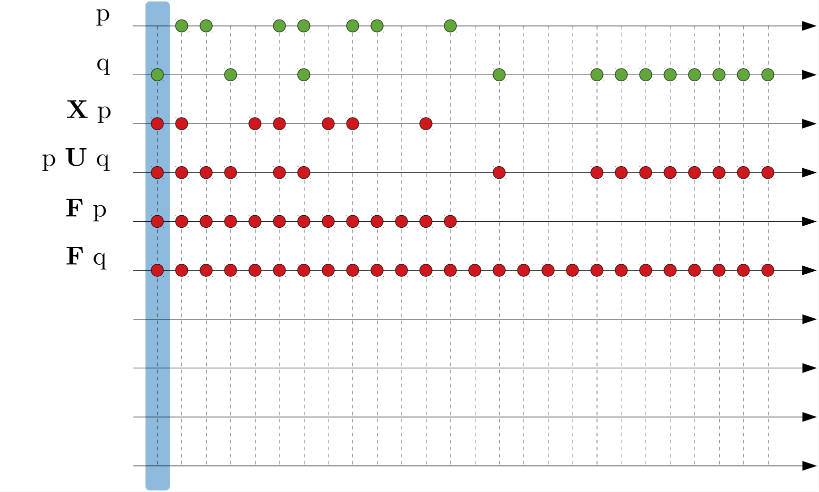



p
q
X p
p U q

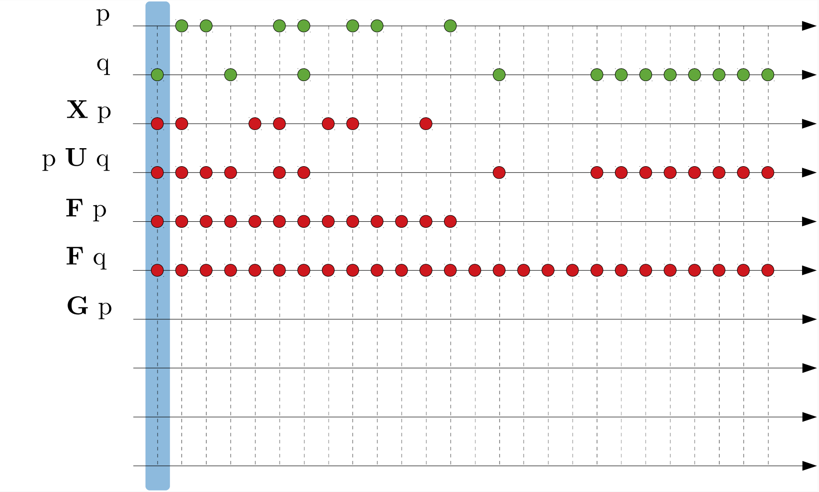
Execution Model & LTL Properties [4/9]



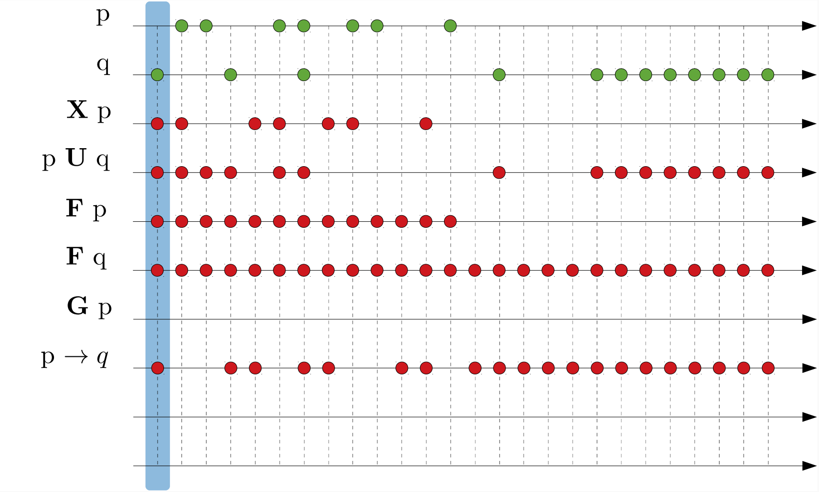
Execution Model & LTL Properties [5/9]



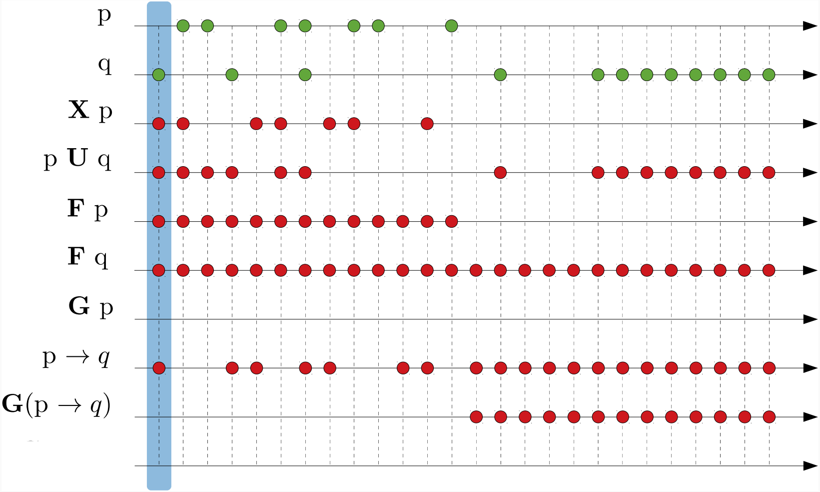
Execution Model & LTL Properties [6/9]



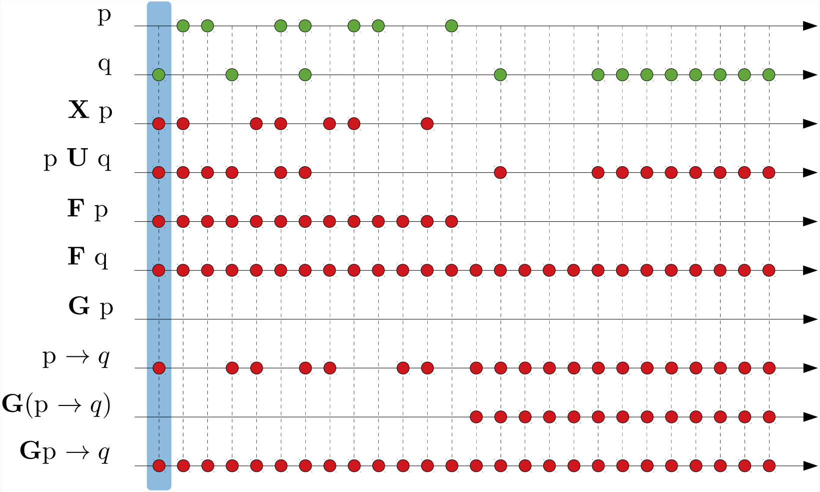
Execution Model & LTL Properties [7/9]



Execution Model & LTL Properties [8/9]



Execution Model & LTL Properties [9/9]



LTL syntax with Spin

- **Grammar**:
  - `ltl ::= opd | ( ltl ) | ltl binop ltl | unop ltl`
- `opd`:
  - `true`, `false`, and user-defined names starting with a lower-case letter
- `unop`:
  - `[]`: globally/always
  - `<>`: finally/eventually
  - `!`: not
  - `X`: next
- `binop`:
  - `U`: until
  - `V`: release
  - `&&`: and
  - `||`: or
  - `->`: implication
  - `<->`: equivalence

remember: $(\varphi V \psi) = !(!\varphi U !\psi)$

Example: LTL model checking [1/2]

Example (foo.pml): verify that `b` is always true.

```
bool b = true;

active proctype main() {
    printf("hello world!\n");
    b = false;
}
```

Example: LTL model checking [1/2]

Example (foo.pml): verify that `b` is always true.

```
bool b = true;

active proctype main() {
    printf("hello world!\n");
    b = false;
}
```

Standard Approach:

- add the LTL formula in `foo.pml`:
  `ltl p1 { [] b }`

Example: LTL model checking [1/2]

Example (foo.pml): verify that `b` is always true.

```
bool b = true;

active proctype main() {
    printf("hello world!\n");
    b = false;
}
```

Standard Approach:

- add the LTL formula in `foo.pml`:
  `ltl p1 { [] b }`
- generate, compile and `run` the verifier:
  ```
  ~$ spin -a foo.pml
  ~$ gcc -o pan pan.c
  ~$ ./pan -a -N p1
  ```

Example: LTL model checking [1/2]

Example (foo.pml): verify that `b` is always true.

```
bool b = true;

active proctype main() {
    printf("hello world!\n");
    b = false;
}
```

Standard Approach:

- add the LTL formula in `foo.pml`:
  ```
  ltl p1 { [] b }
  ```
- generate, compile and `run` the verifier:
  ```
  ~$ spin -a foo.pml
  ~$ gcc -o pan pan.c
  ~$ ./pan -a -N p1
  ```
  or
  ```
  ~$ spin -search -a -ltl p1 foo.pml
  ```

-a: ask the verifier to also check cyclic executions violating a property

Constructs for complex LTL formulas

`_pid`

- unique identifier of a process

Constructs for complex LTL formulas

`_pid`

- unique identifier of a process

`_last`

- pid of the process that performed the last state transition;

Constructs for complex LTL formulas

`_pid`

- unique identifier of a process

`_last`

- pid of the process that performed the last state transition;

`enabled(pid)`

- `true` iff process with identifier `pid` has at least one **executable statement** in its current control state.

Constructs for complex LTL formulas

`_pid`

- unique identifier of a process

`_last`

- pid of the process that performed the last state transition;

`enabled(pid)`

- `true` iff process with identifier `pid` has at least one **executable statement** in its current control state.

Remote References

- allow for inspecting the **local state** of an *active process*:
  - `procname[pid]@label` for **labels**
  - `procname[pid]:varname` for **variables**

Example: (mutual exclusion)

```
ltl p { []! (procname[0]@critical && procname[1]@critical) }
```

Weak Fairness

Weak Fairness: an event E occurs infinitely often.

Example:
every process executes infinitely often

- let R_i be true iff the process i is running
- then a fairrun is s.t.

$$\bigwedge_i \mathbf{GF} R_i$$

- in SPIN:

```
[]<> _last==0 && []<> _last==1 ...
```

Weak fairness is often used as a pre-condition for other properties.

Strong Fairness

Strong Fairness: if an event E_1 occurs infinitely often, then an event E_2 occurs infinitely often.

Example:
if a process is infinitely often ready to execute a statement, then that process runs infinitely often.

- let R_i be true iff the process i is running
- let E_i be true iff the process i can execute a statement
- then a strong_fairrun is s.t.

$$\bigwedge_i (\mathbf{GF} E_i \rightarrow \mathbf{GF} R_i)$$

- in SPIN:

```
[]<> enabled(0) -> []<>_last==0 && ...
```

Example: fairness condition

```
int count;
bool incr;

#define fair ([]<> \
        (incr && _last == 0))

active proctype counter() {
    do
        :: incr ->
            count++
    od
}
active proctype env() {
    do
        :: incr = false
        :: incr = true
    od
}
```

Example:

- Verify the property count reaches the value 10.
- Verify the property above under the fairness condition.

Example: fairness condition

```
int count;
bool incr;

#define fair ([]<> \
        (incr && _last == 0))

active proctype counter() {
    do
        :: incr ->
            count++
    od
}
active proctype env() {
    do
        :: incr = false
        :: incr = true
    od
}
```

Example:

- Verify the property count reaches the value 10.
- Verify the property above under the fairness condition.

Solution:

- `ltl p1 { <> (count == 10) }`
- `ltl p2 { fair -> <> (count == 10) }`

Quiz #1

Q: which properties are verified, and which are not? (Why?)

```
byte x;

active proctype A ()
{
    x = 1;
    do
        :: select(x: 0..10);
    od;
}

ltl p1 { x == 0 }

ltl p2 { x != 0 }

ltl p3 { (x == 0) -> X (x != 0) }

ltl p4 { (x == 0) -> <> (x != 0) }

ltl p5 { [] ((x == 0) -> X (x != 0)) }

ltl p6 { [] ((x == 0) -> <> (x != 0)) }
```

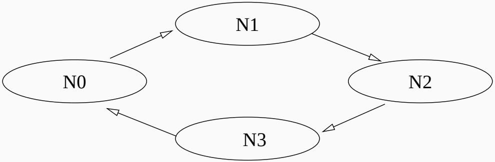
Quiz #1

Q: which properties are verified, and which are not? (Why?)

```
byte x;

active proctype A ()
{
    x = 1;
    do
        :: select(x: 0..10);
    od;
}

ltl p1 { x == 0 }                              // T

ltl p2 { x != 0 }                              // F

ltl p3 { (x == 0) -> X (x != 0) }              // T

ltl p4 { (x == 0) -> <> (x != 0) }             // T

ltl p5 { [] ((x == 0) -> X (x != 0)) }         // F

ltl p6 { [] ((x == 0) -> <> (x != 0)) }        // F
```

Leader Election Problem

- N processes are the nodes of a unidirectional ring network: each process can send messages to its clockwise neighbor and receive messages from its counterclockwise neighbor.
- The requirement is that, eventually, **only one** process will output that it is the **leader**.
- We assume that every process has a **unique id**.
- The leader must have the **highest id**.



Le Lann, Chang, Roberts (LCR) solution

The algorithm:

- Initially, every process passes its identifier to its successor.
- When a process receives an identifier from its predecessor, then:
  - if it is greater than its own, it keeps passing on the identifier.
  - if it is smaller than its own, it discards the identifier.
  - if it is equal to its own identifier, it declares itself leader:
    - the leader communicates to its successor that now it is the leader.
    - after a process relayed the message with the leader id, it exits.

Complexity: at worst, n^2 messages.

Peterson/Dolev, Klawe, Rodeh solution

The algorithm:

- If a process is "active", it compares its identifier with the two counter-clockwise predecessors:
  - if the highest of the three is the counter-clock neighbor, the process proposes the neighbor as leader,
  - otherwise, it becomes a "relay".
- If the process is in "relay" mode, it keeps passing whatever incoming message.

Complexity: at worst, $n \cdot log(n)$ messages.

Exercise 1: Leader Election

```
mtype = { candidate, leader };
chan c[N] = [BUFSIZE] of { mtype, byte };
proctype node(chan prev, next; byte id)
{ ... }

init {
  byte proc, i;
  atomic {
  // TODO: set i random in [0,N]
  ...
  do
  :: proc < N ->
     run node(c[proc],
              c[(proc+1)%N],
              (N+i-proc)%N);
     proc++
  :: else ->
  break
  od
  }
}
```

- Implement a `leader election` algorithm of your choice.
- Verify that there is at most one leader.
- Verify that a leader will emerge.
- Verify that once if a process becomes the leader then it will remain the leader forever.