SPIN: LTL Model Checking*

Patrick Trentin
patrick.trentin@unitn.it
http://disi.unitn.it/trentin

Formal Methods Lab Class, March 16, 2018

Università degli Studi di Trento

(compiled on 17/05/2018 at 12:52)

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le, Alessandra Giordani, Patrick Trentin for FM lab 2005/18

Contents

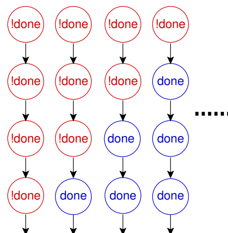
1 Verifying LTL properties with Spin

2 Exercises

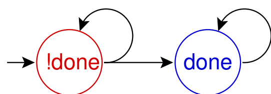
LTL model checking: introduction

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

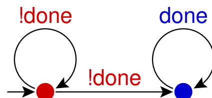
```
bool done = false;
do
    :: done;
    :: else ->
       if
           :: true -> done = true;
           :: true -> skip;
       fi
od;
```



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



or



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

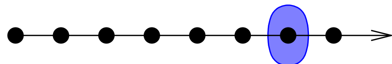
     ($\times$: synchronous product)
2. Check for a possible execution π_i of $A_{\mathcal{M}\times\neg\phi}$:
   - if π_i exists, then it is a violation *(counter-example)* of ϕ in $\mathcal{M}$.
   - otherwise, $\mathcal{M} \models \phi$.

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

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

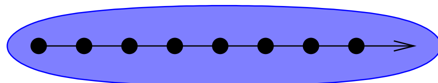
LTL Basics

finally p



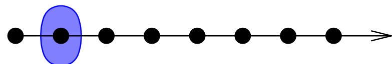
F p

globally p



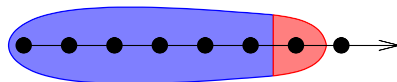
G p

next p



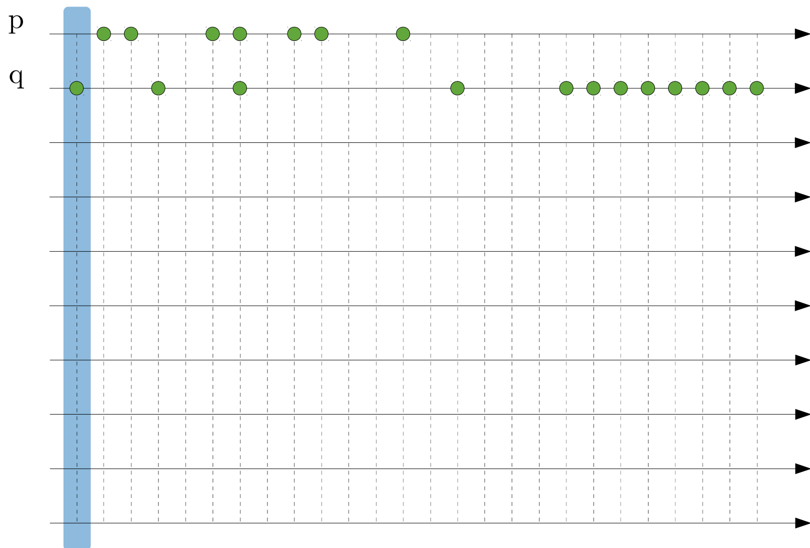
X p

p until q

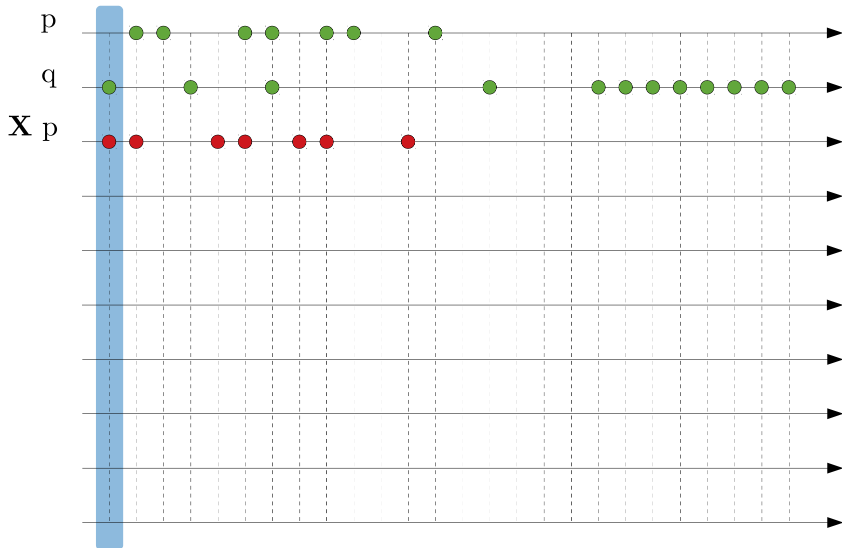


p U q

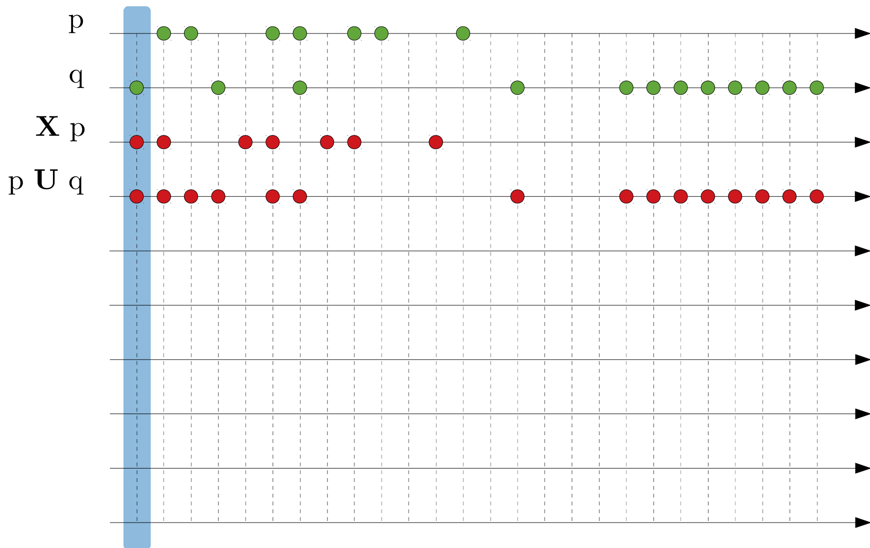
Execution Model & LTL Properties [1/9]



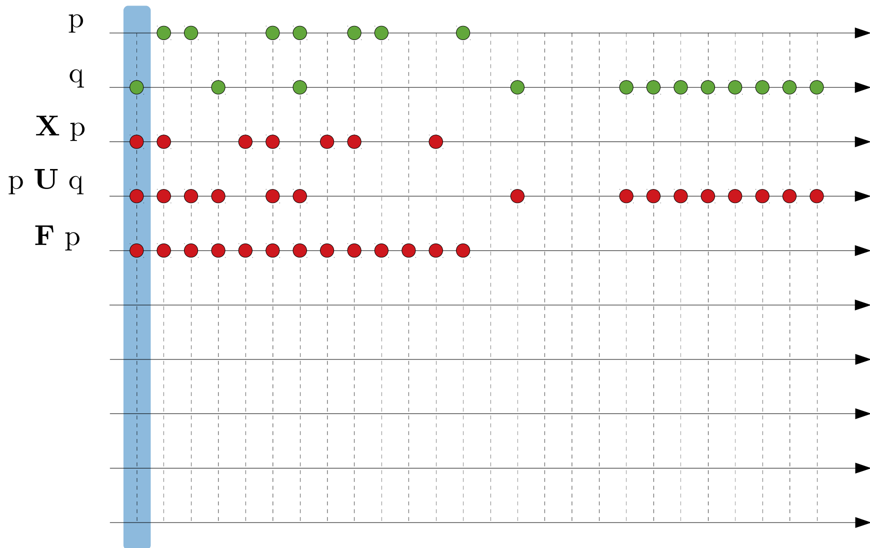
Execution Model & LTL Properties [2/9]



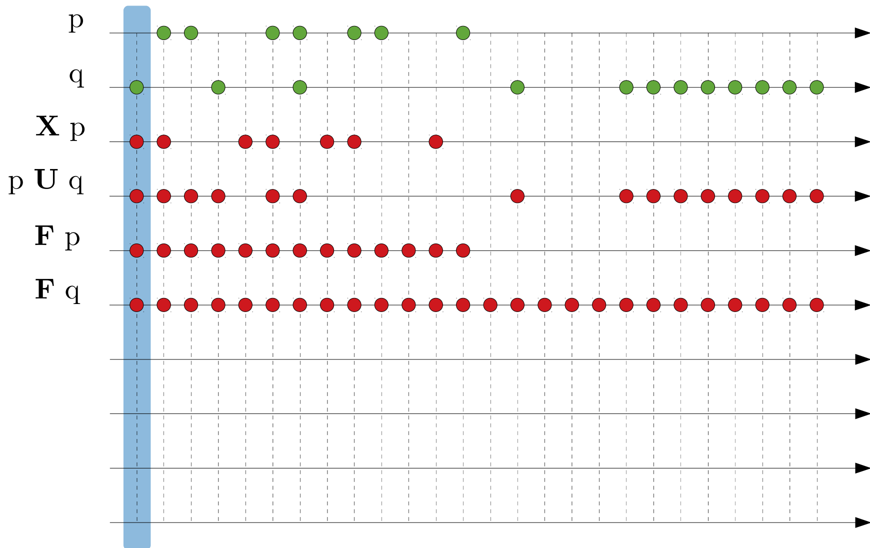
Execution Model & LTL Properties [3/9]



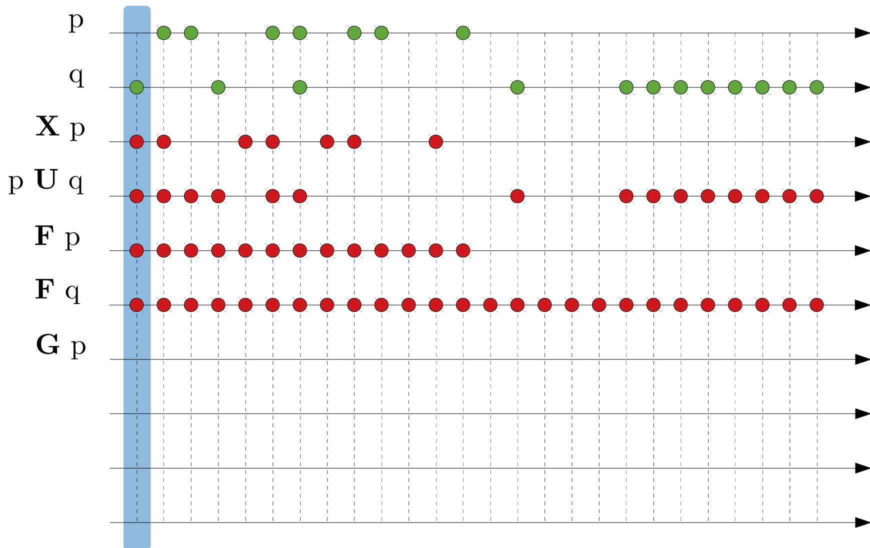
Execution Model & LTL Properties [4/9]



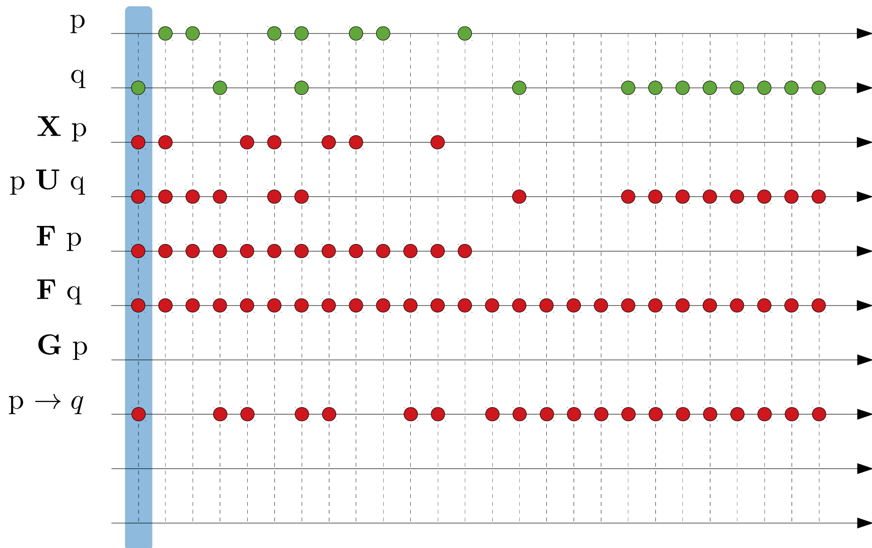
Execution Model & LTL Properties [5/9]



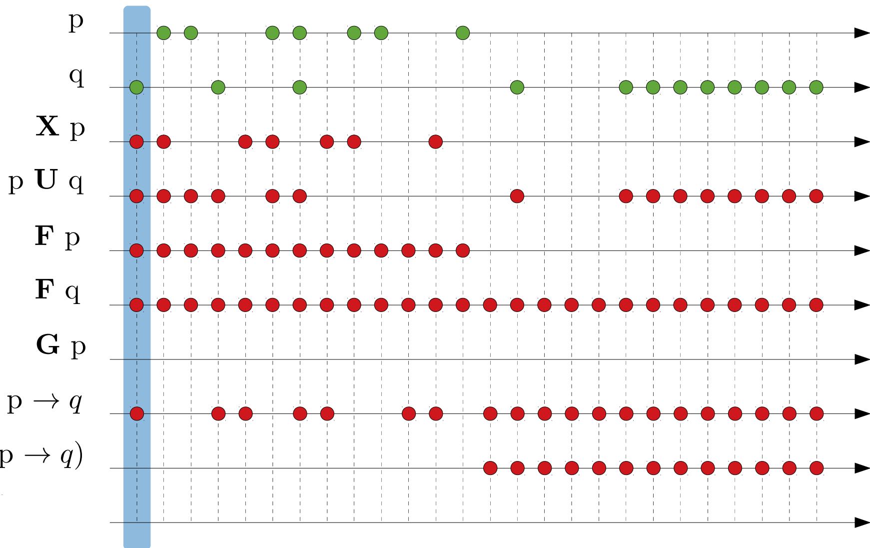
Execution Model & LTL Properties [6/9]



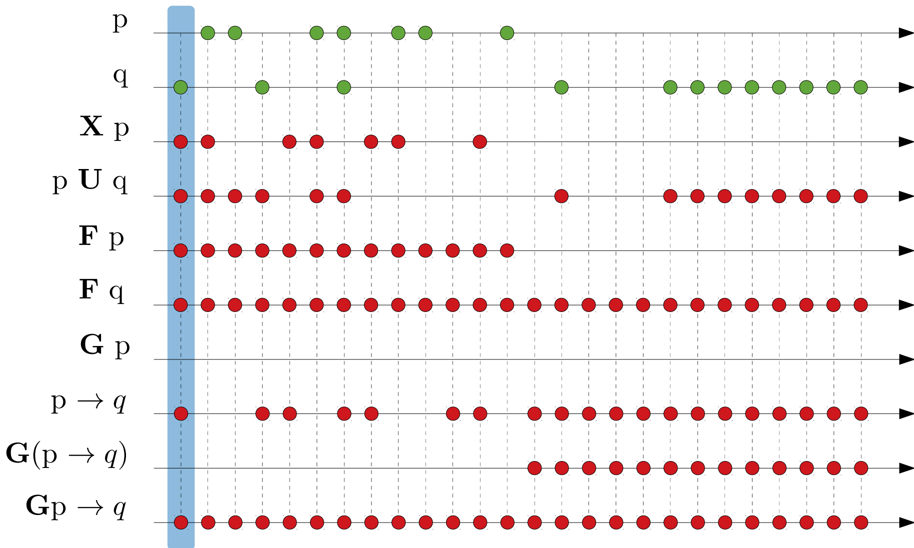
Execution Model & LTL Properties [7/9]



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

Example: LTL model checking [2/2]

Alternative Approach:

- (optional) write some symbol definitions:

```
~$ echo "# define p (b == true)" > foo.aut
```

- generate the never claim to be verified:

```
~$ spin -f '!([] p)' >> foo.aut
```

- generate the verifier:

```
~$ spin -a -N foo.aut -o1 foo.pml
```

  (option `-N file.aut` adds the never claim to the verifier)

- compile and `run` the verifier:

```
~$ gcc -o pan pan.c
~$ ./pan
```

Tip: use the (easier) standard steps!

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 > 9) }`
- `ltl p2 { fair -> <> (count > 9) }`

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
```

Contents

1 Verifying LTL properties with SPIN

2 Exercises

Exercise 1: Leader Election

Verify the following LTL properties on `leader_lcr.pml`:

- eventually, a leader will emerge

$$\mathbf{F}(num_leaders > 0)$$

- there can be at most one leader

$$\mathbf{G}!(num_leaders > 1)$$

- after a process is elected, it will remain leader forever

$$\bigwedge_i \mathbf{G}(elected_i \rightarrow \mathbf{G}\,oneLeader)$$

Exercise 2: Producers/Consumers

Verify the following LTL property on `prodcons.pml`:

- Productions and consumptions must alternate.

$$\mathbf{G}(\bigwedge_{i}(P_i \rightarrow (\bigwedge_{k \neq i} !P_k \mathbf{U} \bigvee_{j} C_j)) \wedge \bigwedge_{j}(C_j \rightarrow (\bigwedge_{k \neq j} !C_k \mathbf{U} \bigvee_{i} P_i)))$$

Exercise 3: Mutual Exclusion

Verify the following LTL properties on `mutex.pml`:

- *mutual exclusion*: there is no reachable state in which more than one process is in the critical section:

$$\mathbf{G}!(\bigvee_{i \neq j}(C_i \wedge C_j))$$

- *progress*: if one process is in the *trying* section, then eventually some process enters the *critical* section:

$$\mathbf{G}(\bigvee_i T_i \rightarrow \mathbf{F} \bigvee_i C_i)$$

- *lockout-freedom*: in a fair path, if a process enters in the *trying* section, then it eventually enters the *critical* section.

$$FAIRRUN \rightarrow \mathbf{G}(\bigwedge_i (T_i \rightarrow \mathbf{F} C_i))$$

Exercise 4: Alternating Bit Protocol

Verify the following LTL properties on `altbit.pml`:

- *response to impulse*: in a fair path, if a message is sent, then it is eventually received.

$$(FAIRRUN \wedge \mathbf{GF}!loss) \rightarrow (\mathbf{G}(sendA \rightarrow \mathbf{F}recA))$$

- *absence of unsolicited response*: if a message is received, then it has been previously sent.

$$\mathbf{F}recA \rightarrow ((\neg recA)\mathbf{U}sentA)$$

- *FIFO*: if B is sent after A, then B is received after A.

$$prec(sendA, sendB) \rightarrow prec(recA, recB)$$

where

$$prec(p, q) := \mathbf{F}q \rightarrow (!q\mathbf{U}p)$$

Exercise 5: Fifo Process

Verify the following LTL properties on `fifo_process.pml`:

- if the fifo is `full`, a write request is not served
- if the fifo is `empty`, a read request is not served
- the counter of fifo elements is always valid (wrt. the size of the FIFO)
- in a *fair run*, if the producer tries to push something on the fifo, then it will eventually succeed
- in a *fair run*, if the consumer tries to pop something from the fifo, then it will eventually succeed

Q1: What happens if the *fair run* requirement is dropped? (Why?)
Q2: What happens if `cc < 10` is replaced with `true`? (Why?)

Exercises Solutions

- will be uploaded on course website within a couple of days
- send me an email if you need help or you just want to propose your own solution for a review

- learning programming languages requires practice: try to come up with your own solutions first!

Optional Exercise: N processes mutual exclusion [1/2]

Model the Black-White Bakery algorithm for N processes:

- before entering the critical section, each process i gets a ticket, defined as a pair $\langle color_i, number_i \rangle$:
  - $color_i$ is set to the current value of a shared bit *color* (of type $\{black, white\}$)
  - $number_i$ is set to a value greater than the number of existing tickets with the same color of its own
- once i has a ticket, it waits until its colored ticket is the lowest, and then it enters the critical section. The order between colored tickets is defined as follows:
  - if two tickets have different colors, the ticket whose color is different from the value of the shared bit color is smaller;
  - if two tickets have the same color, the ticket with the smaller number is smaller;
  - if the tickets of two processes have the same color and the same number then the process with the smaller identifier (*_pid*) enters the critical section first;
- when process i leaves its critical section, it sets the bit *color* to a value which is different from the color of its ticket;

Optional Exercise: N processes mutual exclusion [2/2]

Optional Exercise:

- write a PROMELA model for the Black-White Bakery algorithm for N processes
- check the following properties on $N = 3$:
  - mutual exclusion
  - progress
  - lockout-fredom (for $N = 2$)

  and show that there is no deadlock

Warning: the only awards for successfully solving this exercise are fun, an improved understanding of PROMELA and some confidence that you may be ready to take the first part of the exam. :-)

...I am available for help and hints...
...a solution to this exercise will be provided by the end of the course...