# SPECIFYING AND CHECKING CORRECTNESS PROPERTIES

## with **Linear Temporal Logic** (LTL)

# safety and liveness

# Safety properties…

… express behavior that a system should avoid

**Something bad cannot happen!**

Modeler's perspective

# <u>Safety</u> properties… aka <u>state</u> properties

… express behavior that a system should avoid

Safety properties are about **states**…

- absence of deadlocks
- system **invariants** that hold
  – *globally*: in all system states
  – *locally*: in selected local states, always

Verifier's
(Model Checker's)
perspective

# <u>Liveness</u> properties…

… express behavior that the system should allow

**Something good should happen!**

- *progress, termination, non-starvation, fairness, or (usually) some other kind of positive eventuality*

Modeler's perspective

# Liveness properties… aka path properties

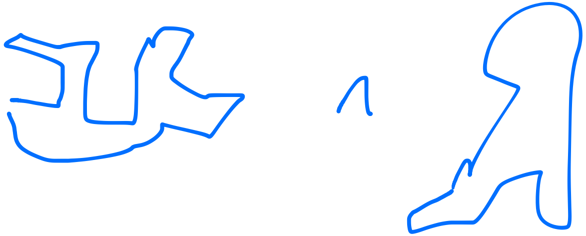
… *express behavior that the system must allow

Liveness properties are about **paths**…
… combining state properties along an execution path to express dependencies between them

Verifier's
(Model Checker's)
perspective

**usually*

# Specifying Properties in Promela/Spin



*State properties*

- assertions
- end-state labels
- Linear Temporal Logic (LTL) properties (only with [])

*Path properties*

- Linear Temporal Logic (LTL) properties (containing <>, U, X)
- never claims
- progress-state labels (desirable cycles)

# Assertions are…

… the simplest way to express safety properties

```
assert(boolean_expression)
```

• State proposition

- always executable
- can be used anywhere a statement can

# Assertion can define a local invariant

*When used inside any process*

```
proctype receiver() {
    ...
    toReceiver ? msg;
    assert(msg != ERROR); // must be true when this process reaches this state
    ...
}
```

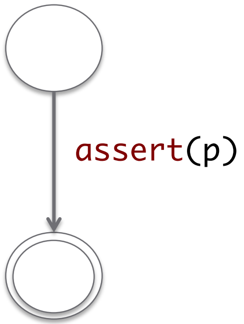
*Violation anytime during verification will produce an error with a corresponding failure trail*

A local **invariant** is a property that always holds true in specific local states

# Assert can be used to check a <u>global</u> <u>invariant</u>

Add a **monitor**, or ***spy***, process to the model, and run

```
active proctype monitor()
{
   assert(p) ;
}
```




`assert(p)` will be enabled at every global state, and checked by verifier

A global **invariant** is a property that holds true in every global state (global state = a state of the product automaton)

# Deadlock: being stuck in an invalid end state

- Automatically detected by default in Spin in Safety verification
  – to disable, don't check *Invalid end states* in SpinRCP

- The closing curly brace of a process is a valid **end state**

# End-state labels define legal termination

To indicate other valid end-states, use state labels prefixed with **end**:

```
end: …
 end1: …
   endstate: …
     end-state: …
```

# A system that stops : Sema1.pml

```
mtype { P, V }

chan s = [0] of { mtype }

active proctype Sema() {
  bit free = 1;

  do
  :: (free == 1) -> s!P; free = 0;
  :: (free == 0) -> s?V; free = 1;
  od;
}

active [3] proctype User() {
  s?P; // enter, no reservation
  critical: skip;
  s!V; // leave
}
```

rendez-vous channel ensures atomicity of semaphore operations, which must be indivisible (atomicity guaranteed when implemented by an OS)

- P: lock that resource
- V: release that resource

- Run *Safety Verification*
  - *Invalid end states*
- *Check that it fails due to invalid end state*
- *But it should be valid...*
- Where do you put the end state in Sema?
  - Before (free == 1)?
  - Before s?V
  - Before s!P

Source: public-spin/Semaphore

Can fix safety check by adding an end state

# Sema2.pml

```
mtype { P, V }

chan s = [0] of { mtype }

active proctype Sema() {
  bit free = 1;

  do
  :: (free == 1) ->
     end: s!P; free = 0;
  :: (free == 0) ->
     s?V; free = 1;
  od;
}

active [3] proctype user() {
  s?P;
  critical: skip;
  s!V;
}
```

end state can't be here because one of the do guards is always executable

this is a valid end state of the semaphore: waiting for another P instruction

- Mark the releasing of the lock as a valid end state. It will pass the verification.

Now each user process enters its critical section only once and terminates: an end state is needed in semaphore implementation to tell Spin this is ok during a safety check

# Sema3.pml - Try at home!

```
mtype { P, V }

chan s = [1] of { mtype };

active proctype Dijkstra() {
  bit free = 1;

  do
  :: (free == 1) ->
     end: s!P; free = 0;
  :: (free == 0) ->
     s?V; free = 1;
  od;
}

active [3] proctype user() {
  s?P;
  critical: skip;
  s!V;
}
```

- Replace rendez-vous channel with a regular channel
  - It's not atomic now. Will not pass.
- Does the verification still pass?
- If not, why?
- Inspect content of chan s: what does it contain?

*See Appendix for more examples on the effect of rendez-vous!*

# Linear Temporal Logic

Can express global correctness properties explicitly and compactly

– both **state** and **path**

- *Logic*: a formal logic in the full mathematical sense
- *Temporal*: operates on ordered execution sequences
- *Linear*: time flows along a straight line, no branching

# Linear Temporal Logic formulas are implicitly *universally quantified over execution paths*

$\forall$

"for all paths (reachable execution traces) of the system"

vs.

"~~there exists a path in the system~~"

# An LTL formula consists of temporal operators…

p: a state proposition (*in Promela: boolean expression that can be checked on a system state, like something we can put in an* assert(…) *statement*) – means p is true at the current state

| | | |
|---|---|---|
| **Always** | []p | p is true throughout |
| **Eventually** | <>p | p is eventually true |
| **Next** | Xp | p is true in the next state |
| **Until** (Strong) | $p_1$ U $p_2$ | $p_1$ stays true until $p_2$ becomes true |

*Main temporal operators*

• Means implication

… *and logical operators*: ||, &&, !, ->, <->

• Negation

-> has a different meaning in LTL; it's implication, not a statement separator

# What is a state proposition?

Any valid PROMELA Boolean expression:

(mutex < 2)

(x > 0 && y < 0)

(len(ch) > 0)

(b == 0)

...

# LTL properties are easy to visualize

***for a single path***

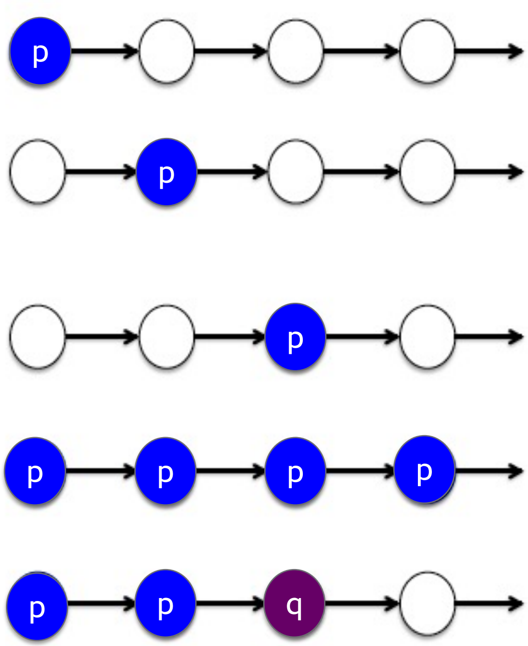
p:   p holds in current state

Xp: p holds in the next state

<>p: p holds eventually
- At least one state p holds true...

[]p: p holds from now on ∀

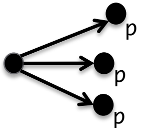
p U q: p holds until q holds

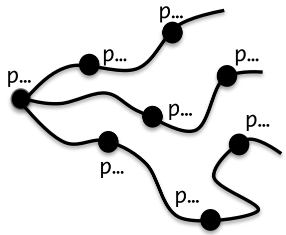


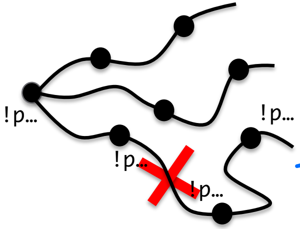

Miranda 2014

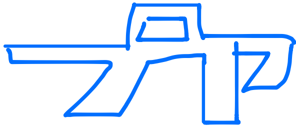
# Simple algorithms can check LTL properties on a product automaton

| | |
|---|---|
| p | check p on the current/initial state |
| Xp | check p holds in all successors of current state |
| []p | find all reachable states from the current state and ensure p holds in all of them |
| <>p | look for a path from the current state on which p is false in every state; if no such path is found, then on every path reachable from the current state, p must be true somewhere, therefore, <>p must be true |

For these two, the worst case would need to explore the whole state space.

# LTL Examples (for a single path)

Let τ ≡ {a} → {b} → {a} → {b} → ...

This means the list is infinite cycles and will not end. That's making []<>(a) true.

- a, b, c are state propositions
- τ is a path (trace/trail)
- {a}, {b}, … indicate the state propositions that are true for each state on a path

{x,y}

means x, y are true, and nothing else in this state

- !b b is not true in the initial state, so satisfied
- a -> Xb
- c -> a The precondition is not satisfied. So this implication is automatically satisfied regardless of post condition
- [](a -> Xb)
- [](b -> Xa)
- <>(a) Evatually a: at least one a will appear in current or future state
- []<>(a) Always eventually a: for every state, there will be at least one a that will appear in current for future state
- [](a -> <>b)
- a U b

A

Caution: In LTL "->" is implication; not a statement separator like it is in Promela

# LTL is defined in the global scope: Normally only global variables of a Promela model canappear in an LTL formula to define state propositions

*If you want to refer to a local variable….*

can’t refer to x here

```
bit gx;

ltl p { []((gx == 0) -> <>(gx == 1)) }

active proctype X() {
    bit x;

     d_step { x = 0; gx = x; }
     do
     :: x == 1; d_step { x--; gx = x; }
     :: x == 0; d_step { x++; gx = x; }
     od;
}
```

Define a global var *gx* that tracks x, and copy x to *gx* whenever x is updated

See “Remote References” slides in Appendix for ways of accessing local variables or local symbolic states in a model or in an LTL

# Example: Sema4.pml

```
mtype { P, V }
chan s = [0] of { mtype }

byte count = 0; // counting users in critical section
bool wantIt[2] = 0; // user i wants the semaphore
bool getIt[2] = 0; // user i got the semamphore

active proctype Sema() {
  do
  ::  s!P; s?V;
  od
}

proctype User(byte i) {
  do
  :: wantIt[i] = 1;
     s?P ->
     count++;
     wantIt[i] = 0; getIt[i] = 1; // critical section
     count--;
     s!V; getIt[i] = 0
  od;
}


init {
  run User(0);
  run User(1);
}
```

- Let's define some *observable global variables* for the simplified version of the Semaphore example

- Define the mutex property as an LTL
- Define a progress property for each user as an LTL:
  - always, if a user process wants the semaphore, it eventually gets the semaphore

*Note that* **observation variables** *are not used to control the behavior in any way!*

`count is like the mutex var in the mutex example`

`Remember: For bool and bit vars: 1 == true, 0 == false`

Source: public-spin/Semaphore

# Sema4.pml

```
mtype { P, V }
chan s = [0] of { mtype }

byte count = 0; // counting users in critical section
bool wantIt[2] = 0; // user wants the semaphore
bool getIt[2] = 0; // user got the semamphore

active proctype Sema() {
  do
  ::   s!P; s?V;
  od
}

proctype User(byte i) {
  do
  :: wantIt[i] = 1;
     s?P ->
     count++;
     wantIt[i] = 0; getIt[i] = 1;
     count--;
     s!V; getIt[i] = 0
  od;
}


init {
  run User(0);
  run User(1);
}

/*
   Verification -> Liveness, Acceptance cycles,  Apply never claim
   Specify name of ltl
*/
ltl mutex { [](count < 2) }
ltl wantItGetIt0 { [](wantIt[0] -> <>getIt[0]) }
ltl wantItGetIt1 { [](wantIt[1] -> <>getIt[1]) }
```

"You want it, you get it." :)

- *Verification -> Liveness*
  - *Acceptance cycles*
  - *Apply never claim*
  - Turn off *Use partial order reduction (Advanced Options)*
  - Specify *in-model LTL formula*
    - mutex
    - wantItGetIt0
    - wantItGetIt1

- Which ones fail, which ones pass?
  - Sema4 ⊨ mutex?
  - Sema4 ⊨ wantItGetIt0?
  - Sema4 ⊨ wantItGetIt1?

- Check console output
- Check failure trace by running a guided simulation
- Notice <<<<START OF CYCLE>>>>

# Sema4.pml

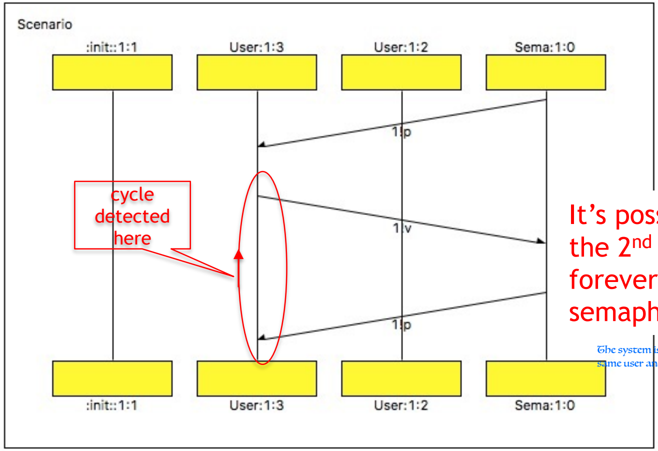
Verification result:
pan: ltl formula wantItGetIt1
pan:1: acceptance cycle (at depth 11)
pan: wrote Sema4.pml.trail

Sema4 ⊭ wantItGetIt1

State-vector 60 byte, depth reached 30, errors: 1
14 states, stored
0 states, matched
14 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):

| | |
|---|---|
| 0.001 | equivalent memory usage for states (stored*(State-vector + ( |
| 0.287 | actual memory usage for states |
| 128.000 | memory used for hash table (-w24) |
| 0.534 | memory used for DFS stack (-m10000) |
| 128.730 | total actual memory usage |




It’s possible for the 2nd user to forever *get* the semaphore

The system is not fair. The semaphore is always given to the same user and not the other.

# Frequently used LTL formulae

S: state
P: path

| Formula | Pronounced | Interpretation | Type |
|---|---|---|---|
| []p | always p | invariance | S |
| <>p | eventually p | guarantee | P |
| p -> <>q | p implies eventually q | response | P |
| []<>p | always eventually p (p holds *infinitely often*) | recurrence (progress) | P |
| [](p -> <>q) | always, p implies eventually q | recurrent response | P |
| <>[]p | eventually always p (p becomes *persistent*) | stability or non-progress | P |
| !q U (p && !q) | p (strictly) precedes q (if q ever happens) | (strict) precedence | P |
| <>p -> <>q<br>p may before q, q may before p | eventually p implies eventually q | correlation | P |

If p is true for the first time, q must not hold true at the same time.

Other common LTL patterns: http://people.cis.ksu.edu/~dwyer/SPAT/ltl.html
https://matthewbdwyer.github.io/psp/patterns.html

# Basic LTL equivalence rules

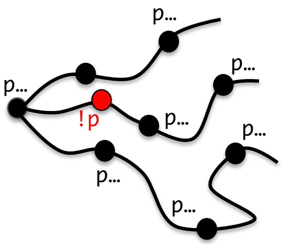
| | | |
|---|---|---|
| ![]p | ⇔ | <>!p |
| !<>p | ⇔ | []!p |
| [](p && q) | ⇔ | []p && []q |
| <>(p || q) | ⇔ | <>p || <>q |

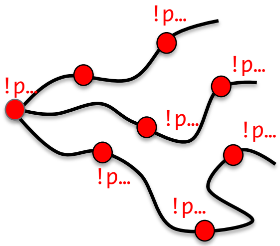
# False friends

| | | |
|---|---|---|
| ![]p | $\not\Leftrightarrow$ | []!p |
| !<>p | $\not\Leftrightarrow$ | <>!p |

# Some false LTL rules

| | | |
|---|---|---|
| ~~! []p~~ | ~~⇔~~ | ~~[] !p~~ |
| ~~! <>p~~ | ~~⇔~~ | ~~<> !p~~ |




! []p ← [] !p

# Extreme Caution!

$\vDash$: **satisfies**

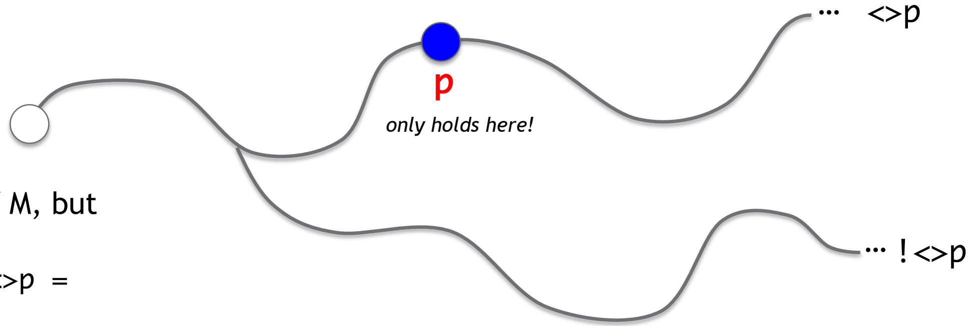
## $M \nvDash \varphi$ does not imply $M \vDash\ !\varphi$

**(it's possible that neither holds)**

### *Reason: implicit universal quantification*

Suppose $\varphi$ = <>p such that

p holds somewhere in some trace of M, but
it holds nowhere in other traces
Then neither <>p nor its negation !<>p =
[]!p will hold for M



# Exercises

Try to work this out in a group of 2-3: 10 mins

*Express each requirement as an LTL formula based on these system state propositions*

- reqS: true if request sent; false otherwise
- reqR: true if request received; false otherwise
- ackR: true if acknowledgement received; false otherwise

*Requirements*

- the request is eventually sent, and once sent, it is always eventually received
- after receiving an acknowledgement, another acknowledgement cannot be received until after a new request has been sent

# Exercises: (non-unique) solutions
# *because we don't know the whole context*

- reqS: request sent
- reqR: request received
- ackR: acknowledgement received

- the request is eventually sent, and once sent, it is always eventually received

```
(<>reqS)
```

# Exercises: (non-unique) solutions

- reqS: request sent
- reqR: request received
- ackR: acknowledgement received

- the request is eventually sent, and once sent, it is always eventually received
- the request is eventually sent, and always, once sent, it is always eventually

```
(<>reqS) && ([](reqS -> <>reqR)) // recurrent response
```

# Exercises: (non-unique) solutions

- reqS: request sent
- reqR: request received
- ackR: acknowledgement received

- the request is eventually sent, and once sent, it is always eventually received

```
(<>reqS) && ([](reqS -> <>reqR)) // recurrent response
```

- after receiving an acknowledgement, another acknowledgement cannot be received until after a new request has been sent

```
ackR -> X(!ackR U … )
```

# Exercises: (non-unique) solutions

- reqS: request sent
- reqR: request received
- ackR: acknowledgement received

- the request is eventually sent, and once sent, it is always eventually received

```
(<>reqS) && ([](reqS -> <>reqR)) // recurrent response
```

- after receiving an acknowledgement, another acknowledgement cannot be received until after a new request has been sent

```
ackR -> X(!ackR U (reqS && !ackR)) // strict precedence
```

# Exercises: (non-unique) solutions

- reqS: request sent
- reqR: request received
- ackR: acknowledgement received

- the request is eventually sent, and once sent, it is always eventually received

```
(<>reqS) && ([](reqS -> <>reqR)) // recurrent response
```

- after receiving an acknowledgement, another acknowledgement cannot be received until after a new request has been sent (… decide that this must be true always)

```
[](ackR -> X(!ackR U (reqS && !ackR))) // strict precedence
```

# Start Working on L4

Review the instructions, due dates, submission limits, and partnering rules both on Canvas and on Vocareum carefully before starting!

If you activate Vocareum solo without specifying a partner, you will need to finish L4 alone!

# Appendix

# Promela Quick Reference

http://spinroot.com/spin/Man/Quick.html

# Example: checking for pure atomicity

Suppose we want to check that none of the atomic clauses in our model are ever blocked (i.e. pure atomicity).

1. Add a global bit variable:

```
bit aflag;
```

2. Change all atomic clauses to:

```
atomic {
   stat₁;
   aflag=1;
   stat₂

   ...

   statₙ
   aflag=0;
}
```

3. Check that aflag is always 0.

```
[]!aflag
```

e.g.

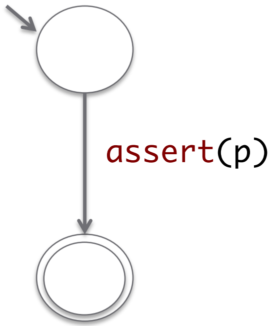
```
active process monitor {
   assert(!aflag);
}
```

Ruys, 2002

# Global invariants can be defined in different ways, some more efficient than others

```
active proctype monitor()
{
    assert(p) ;
}
```

```
active proctype monitor()
{
    do
    :: assert(p)
    od
}
```




*Same effect, but more efficient since it has a single state*

# More on deadlocks…

- Automatically detected by default in Spin in most verification modes (can be turned off – don’t check *Invalid end states* in SpinRCP *Verification*)

- Whenever Spin reaches a global state with no enabled transitions and that are not valid **end states**, it will report an error

- The PC location just before the closing curly brace of a **proctype** is always a valid **end state** for that process (will not lead to reporting of a deadlock)

- **timeout** statements may disable deadlock states
  - *be careful about not masking possible deadlock behavior by a global timeout!*

# More on end states: end-state labels define legal termination

If a process counter (PC) location is labeled as an end state, a state ending at that location will be treated as an intentional and valid end state (deadlock will not be reported even if execution is not possible out of that state)

End-state labels are prefixed with the keyword end and must be unique in each process

- E.g., `end`, `end1`, `endstate`, `end-state`

Source: public-spin/Semaphore

# End-state example: Disjktra's semaphore: an infinitely running system (never reaches an end state)

```
mtype { P, V }

chan s = [0] of { mtype }

active proctype Sema() {
  bit free = 1;
  do
  :: (free) -> s!P; free = 0;
  :: (!free) -> s?V; free = 1;
  od;
}

active [3] proctype user() {
  do
  :: s?P;   // enter
     critical: skip;
     s!V; // leave
  od;
}
```

rendez-vous channel ensures atomicity of semaphore operations, which must be indivisible

3 users competing for the same semaphore

Since the users keep requesting the semaphore, no invalid end states found in Spin verification

Run *Random Simulation*, see that P and V never gets interleaved and the system runs indefinitely

Source: public-spin/Semaphore

# More on rendez-vous: rendez-vous reduces interleaving

## SemaRVOk.pml

```
#define MAX 0 // this works
#define USERS 3

mtype { P, V }
byte count = 0;

chan s = [MAX] of { mtype }

active proctype Sema() {
      do
      :: s?P; s?V;
      od;
}

active [USERS] proctype User() {
      do
      ::    s!P;  // enter
            critical:  count++;
            leaving:   count--
            s!V;  // leave
      od;
}

active proctype mutex () {
   assert(count <= 1);
}
```

SemaRVOk ⊨ mutex

## SemaRegNotOk.pml

```
#define MAX 2 // this doesn't work
#define USERS 3

mtype { P, V }
byte count = 0;

chan s = [MAX] of { mtype }

active proctype Sema() {
      do
      :: s?P; s?V;
      od;
}

active [USERS] proctype User() {
      do
      ::    s!P;  // enter
            critical:  count++;
            leaving:   count--
            s!V;     // leave
      od;
}

active proctype mutex () {
   assert(count <= 1);
}
```

SemaRegNotOk ⊭ mutex

*Verification -> Safety -> Assertion violations | Simulation -> Random*

Source: public-spin/Semaphore

# More on rendez-vous: rendez-vous reduces interleaving

## SemaRVOk.pml

```
#define MAX 0 // this works
#define USERS 3

mtype { P, V }
byte count = 0;

chan s = [MAX] of { mtype }

active proctype Sema() {
      do
      :: s?P; s?V;
      od;
}

active [USERS] proctype User() {
      do
      ::    s!P;  // enter
            critical:  count++;
            leaving:   count--
            s!V;  // leave
      od;
}

active proctype mutex () {
   assert(count <= 1);
}
```

SemaRVOk $\models$ mutex

## SemaRegNotOk.pml

```
#define MAX 2 // this doesn't work
#define USERS 3

mtype { P, V }
byte count = 0;

chan s = [MAX] of { mtype }

active proctype Sema() {
      do
      :: s?P; s?V;
      od;
}

active [USERS] proctype User() {
      do
      ::    s!P;  // enter
            critical:  count++;
            leaving:   count--
            s!V;     // leave
      od;
}

active proctype mutex () {
   assert(count <= 1);
}
```

S & V ops no longer indivisible

SemaRegNotOk $\not\models$ mutex

*Verification -> Safety -> Assertion violations | Simulation -> Random*

Source: public-spin/Semaphore

# Rendez-vous reduces interleaving

## SemaRVOk.pml

```
#define MAX 0 /* this works */
#define USERS 3

mtype { p, v };
byte count = 0;

chan s = [MAX] of { mtype };

active proctype Sema()
{
	do
	:: s?p; s?v;
	od
}

active [USERS] proctype User()
{
	do
	::	s!p;		// enter
		critical:	count++;
		leaving:	count--
		s!v;		// leave
	od
}

active proctype mutex () {
   assert(count <= 1)
}
```

SemaRVOk $\models$ mutex

## SemaRegNotOk.pml

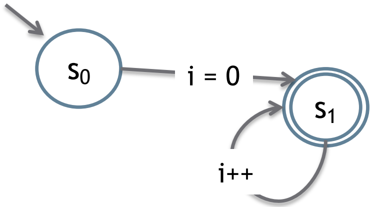
```
#define MAX 2 /* this doesn't work */
#define USERS 3

mtype { p, v };
byte count = 0;

chan s = [MAX] of { mtype };

active proctype Sema()
{
	do
	:: s?p; s?v;
	od
}

active [USERS] proctype User()
{
	do
	::	s!p;		// enter
		critical:	count++;
		leaving:	count--
		s!v;		// leave
	od
}

active proctype mutex () {
   assert(count <= 1)
}
```

SemaRegNotOk $\not\models$ mutex

*Verification -> Safety -> Assertion violations | Simulation -> Random*

# A clarification on *global* vs. *local* state

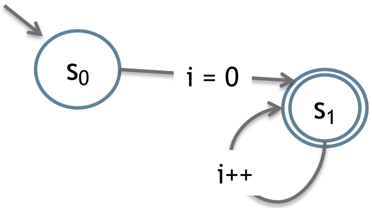
# What program code could this FSA represent




i is one-byte unsigned integer

# What is the complete state space of this FSA

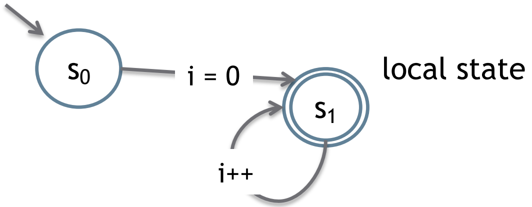
```
active proctype forever() {
    byte i;
    i = 0;
s1:
    do
    :: i++
    od
}
```



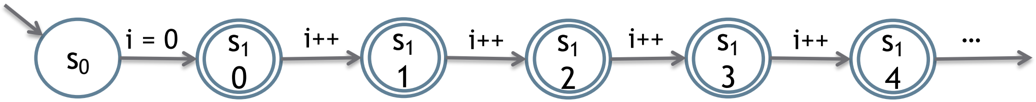

i is one byte unsigned integer

How many global states?
(in Spin)

# What is the complete state space of this FSA




i is one byte unsigned integer




global state

257 global states?

# Local state is different from global state

*Process state as defined in a model is local*: represents execution/process counter, or node-label of underlying FSA

*System state is global* and comprises:

- identities of all active processes
- local state of all active processes
- values of all local variables of all active processes
- contents of all channels
- values of all global variables

# LTL interpretation examples

| Formula | Interpretation |
|---|---|
| `[]p` | p is invariantly true |
| `<>[]!p` | p eventually becomes invariantly false |
| `[]<>!p` | p always eventually becomes false, at least once more |
| `[](q -> <>!p)` | q always implies eventually p becomes false |
| `[](q -> !p)` | q always implies p is false in the same state |

LTL operator precedence: unary *over* binary

*Always use brackets to be prudent!*

# Temporal illusions

*Be careful!*

| | |
|---|---|
| `p -> q` | p implies q <u>only</u> in initial state |
| `[](p -> q)` | p implies q in <u>every</u> state, locally for that state |
| `[](p -> <>q)` | ok, but p and q may be true together (no strict ordering) |
| `[](p -> X<>q)` | q becomes true <u>only</u> after p becomes true (strict ordering, *but* trivially true if p is never true)<br>*or did you mean this?* |
| `(<>p) && ([](p -> X<>q))` | p must eventually be true <u>and</u> q becomes true after p becomes true |

# A word about recurrent response

`[](p -> <>q)`

Problem
- May trivially hold if `p` is never true
- May need to check that:
  - (a) `p` can be true in all traces (eventually `p`) – *may be too strong*
  - (b) `p` can be true along *some* trace

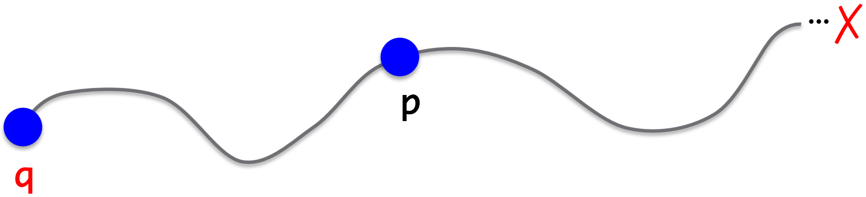
Solution
- (a) check pre-condition separately by checking `<>p` holds
- (b) check pre-condition separately by `[]!p`, but for failure
  - If `[]!p` fails, then the counterexample is proof that `p` holds somewhere for *some* trace

*Remember*
- Checking `![]!p = <>p` holds for a model is not the same as checking `[]!p` fails for that model
- Both `![]!p` and `[]!p` may fail for a model simultaneously because of implicit universal quantification

# Recurrent response

Always Eventually

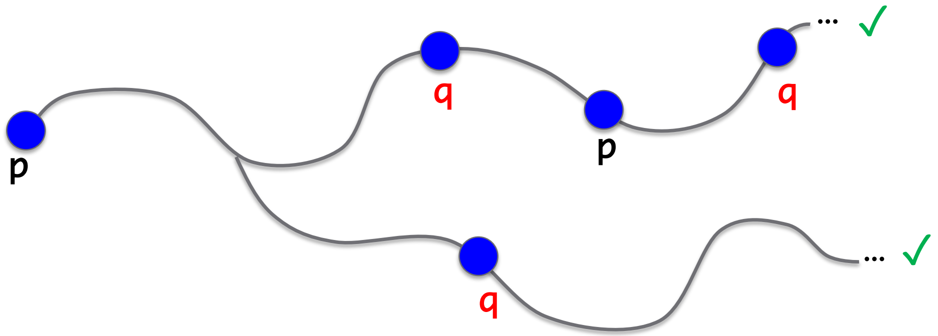
```
[](p -> <>q)
```



# Recurrent response

Always Eventually

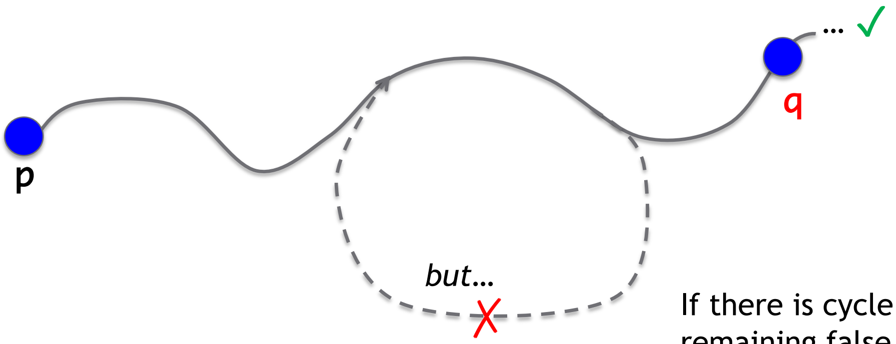
`[](p -> <>q)`



# Recurrent response

Always                Eventually

[](p -> <>q)

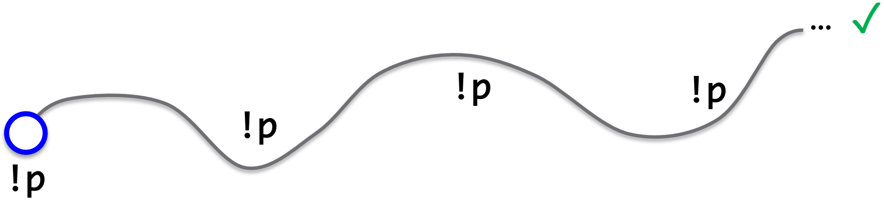



If there is cycle with q remaining false, then it's a no go

# Recurrent response

Always　　　Eventually

`[](p -> <>q)`



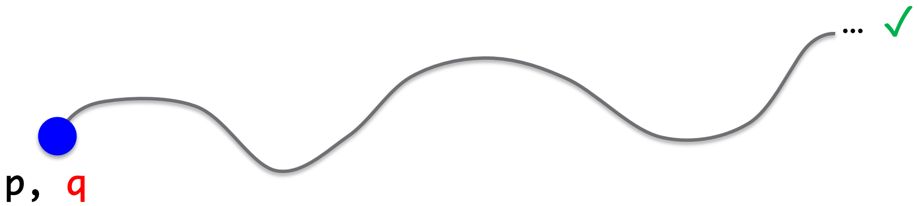

*Ok because…*

`[]!p`

# Recurrent response

Always         Eventually

`[](p -> <>q)`



# Safety/state properties: mechanisms

- end-state labels

```
proctype Sema()
   endSema:
   do
    :: s!p;
    :: s?v;
    od
}
```

- assert

```
active proctype always() {
    assert(count < 2) ;
}
```

- monitor processes

- LTL: exclusive use of temporal operator []

```
ltl alwaysLessThanTwo { [](count < 2) }
```

# Liveness/path properties: mechanisms

- LTL: inclusion of temporal operators <>, X, U

```
ltl alwaysEventuallyZero { []<>(count == 2) }
```

- progress labels (not covered)
- never claims (not covered)

# Liveness properties -- Progress: the system will move to a productive state

```
ltl willProgress { <>(mutex > 0) }
```

```
bit x, y;
byte mutex = 0;

active proctype A() {
    x = 1;
    y == 0;
    mutex++;
// A in critical section
    mutex--;
    x = 0;
}
```

```
active proctype B() {
    y = 1;
    x == 0;
    mutex++;
// B in critical section
    mutex--;
    y = 0;
}
```

# Livness properties - Termination: the system will eventually stop at a valid end state

```
bool done = false;

ltl stops { <>(done) }

active proctype Q() {
    int i = 0;
    do
    :: i < MAX -> i++;
    :: i < MAX -> break;
    :: i == MAX -> break;
    od
    done = true;
}
```

# Specifying termination with LTL (liveness)

```
bool ok = false;

ltl p { <>(ok) } // fails

active proctype Q() {
    byte i = 0;
    byte MAX = 10;
    do
    :: i < MAX -> i++;
    :: i <= MAX -> i--;
    :: i == MAX -> break;
    od
    ok = true;
}
```

```
Verification result:

warning: only one claim defined, -N ignored
pan:1: acceptance cycle (at depth 18)
pan: wrote Termination.pml.trail

(Spin Version 6.3.2 -- 17 May 2014)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             + (p)
        assertion violations    + (if within scope of
claim)
        acceptance   cycles     + (fairness enabled)
        invalid end states      - (disabled by never
claim)

State-vector 28 byte, depth reached 24, errors: 1
         13 states, stored (16 visited)
          1 states, matched
         17 transitions (= visited+matched)
          0 atomic steps
```

Source: public-spin/ABP

# AltBit4.pml

```
mtype = { msg, ack }
chan  to_sndr = [2] of { mtype, bit }
chan  to_rcvr = [2] of { mtype, bit }
bit msgSent[2] = 0; // for LTL
bit ackReceived[2] = 0; // for LTL


active proctype Sender() {
  bit seq_out = 0;
  bit seq_in = 0;
  // obtain first message
  do
  :: if
     :: to_rcvr!msg(seq_out);
     // send message
     :: skip; // or loose it
     fi;
     msgSent[seq_out] = 1;
     msgSent[seq_out] = 0;
     to_sndr?ack(seq_in);
     ackReceived[seq_in] = 1;
     ackReceived[seq_in] = 0;
     if
     :: seq_in == seq_out ->
        // obtain new message
        seq_out = 1 - seq_out;
     :: else; // retransmit same message
     fi;
  od;
}
```

```
ltl msg0Sent { <>msgSent[0] }
ltl msg1Sent { <>msgSent[1] }
ltl recover0 { [](msgSent[0] -> <>ackReceived[0]) }
ltl recover1 { [](msgSent[1] -> <>ackReceived[1]) }
```

```
active proctype Receiver() {
  bit seq_in = 1;  // important
  do
  :: if
     :: to_rcvr?msg(seq_in);  // receive msg
     :: timeout; // recover from msg loss
     fi;
     if
     :: to_sndr!ack(seq_in); // send ack
     :: skip; // or loose it
     fi;
  od;
}
```

- *Verification* of each LTL
  - *Liveness*
  - *Acceptance cycles*
  - *Apply never claim*
  - Turn off *Use partial order reduction (Advanced Options)*
- Which ones fail, which ones pass?
- Check console output
- Check failure trace

Source: public-spin/ABP

# AltBit4.pml

```
mtype = { msg, ack }
chan     to_sndr = [2] of { mtype, bit }
chan     to_rcvr = [2] of { mtype, bit }
bit msgSent[2] = 0; // for LTL
bit ackReceived[2] = 0; // for LTL

ltl msg0Sent { <>msgSent[0] }
ltl msg1Sent { <>msgSent[1] }
ltl recover0 { [](msgSent[0] -> <>ackReceived[0]) }
ltl recover1 { [](msgSent[1] -> <>ackReceived[1]) }

active proctype Sender() {
  bit seq_out = 0;
  bit seq_in = 0;
  // obtain first message
  do
     :: if
        :: to_rcvr!msg(seq_out);  // send message
        :: skip; // or loose it
        fi;
        msgSent[seq_out] = 1;
        msgSent[seq_out] = 0;
        to_sndr?ack(seq_in);
        ackReceived[seq_in] = 1;
        ackReceived[seq_in] = 0;
        if
        :: seq_in == seq_out ->
          // obtain new message
          seq_out = 1 - seq_out;
        :: else; // retransmit same message
        fi;
  od;
}

active proctype Receiver() {
  bit seq_in = 1; // important
  do
  :: if
     :: to_rcvr?msg(seq_in);  // receive msg
     :: timeout; // recover from msg loss
     fi;
     if
     :: to_sndr!ack(seq_in); // send ack
     :: skip; // or loose it
     fi
  od;
}
```

- *Verification* for each LTL
  - *Liveness*
  - *Acceptance cycles*
  - *Apply never claim*

- AltBit4 ⊨ msg0Sent
- AltBit4 ⊭ msg1Sent
- AltBit4 ⊭ recover0
- AltBit4 ⊭ recover1

- Must limit message losses

Source: public-spin/ABP

# AltBit5.pml

- AltBit5 ⊨ recover0
- AltBit5 ⊨ recover1
- AltBit5 ⊨ maxLoss
- AltBit5 ⊭ FAIL_noLoss
- AltBit5 ⊭ FAIL_noLossMax

```
…

byte lostMsg = 0; // count lost msg
byte lostAck = 0; // count lost ack
byte MAXLOST = 3; // limit max no. of losses

ltl recover0 { [](msgSent[0] -> <>ackReceived[0]) }
ltl revover1 { [](msgSent[1] -> <>ackReceived[1]) }
ltl maxLoss { [](lostMsg <= MAXLOST && lostAck <= MAXLOST) }
// these last two should all fail - both channels are lossy up to MAXLOST
ltl FAIL_noLoss { [](lostMsg == 0 && lostAck == 0) }
ltl FAIL_noLossMax { [](lostMsg < MAXLOST && lostAck < MAXLOST) }
```

## Limit losses!

```
active proctype Sender() {
  bit seq_out = 0;
  bit seq_in = 0;
  // obtain first message
  do
  :: if
     :: (lostMsg > MAXLOST) -> atomic {
          to_rcvr!msg(seq_out);
          lostMsg = 0;
        } // definitely send message
     :: (lostMsg <= MAXLOST) ->
        if
        :: atomic { to_rcvr!msg(seq_out);
             lostMsg = 0;
           } // send message
        :: lostMsg++ ; // or loose it
        fi;
     fi;
     msgSent[seq_out] = 1;
     msgSent[seq_out] = 0;
     to_sndr?ack(seq_in);
     ackReceived[seq_in] = 1;
     ackReceived[seq_in] = 0;
     if
     :: seq_in == seq_out ->
        // obtain new message
        seq_out = 1 - seq_out;
     :: else; // retransmit same message
     fi;
  od;
}
```

```
active proctype Receiver() {
  bit seq_in = 1; // important
  do
  :: if
     :: to_rcvr?msg(seq_in);  // receive msg
     :: timeout; // recover from msg loss
     fi;
     if
     :: (lostAck > MAXLOST) -> atomic {
          to_sndr!ack(seq_in);
          lostAck = 0;
        } // send ack
     :: (lostAck <= MAXLOST) ->
        if
        :: atomic { to_sndr!ack(seq_in) ->
             lostAck = 0;
           }
        :: lostAck++ // or loose it
        fi;
     fi;
  od;
}
```

# Liveness properties -- Non-starvation: the system will never indefinitely deny access to a resource (it will be fair to all users of resource)

```
bool accessed[2], reserved[2];
#define reserveResource(i) …
#define useResource(i) …

ltl guaranteeAccess0 { [](reserved[0] -> <>accessed[0] }
ltl guaranteeAccess1 { [](reserved[1] -> <>accessed[1] }

active[2] proctype Client(int i) {
   accessed[i] = false; requested[i] = false;
   do // repeatedly access resource
   :: reserveResource(i); reserved[i] = true;
      useResource(i); accessed[i] = true;
      reserved[i] = false; accessed[i] = false;
   od;
}
…
```

# Liveness properties (e.g., non-starvation) may rely on fairness assumptions

**Weak** form: If a statement becomes enabled and stays enabled, it will eventually be executed

– *if a process waits at a state in which in can make progress, it eventually will*

**Strong** form: If a statement is infinitely often enabled, it will eventually be executed

– *if a process constantly revisits a state in which in can make progress, it eventually will*

*We can force Spin to <u>assume</u> **weak fairness** when checking liveness/path properties*

# Some more LTL rules

<table>
<tr><td>![]p</td><td>⇔</td><td><>!p</td></tr>
<tr><td>!<>p</td><td>⇔</td><td>[]!p</td></tr>
<tr><td>[](p && q)</td><td>⇔</td><td>[]p && []q</td></tr>
<tr><td><>(p || q)</td><td>⇔</td><td><>p || <>q</td></tr>
<tr><td>p U (q || r)</td><td>⇔</td><td>(p U q) || (p U r)</td></tr>
<tr><td>(p && q) U r</td><td>⇔</td><td>(p U r) && (q U r)</td></tr>
<tr><td>[]<>(p || q)</td><td>⇔</td><td>([]<>p) || ([]<>q)</td></tr>
<tr><td><>[](p && q)</td><td>⇔</td><td>(<>[]p) && (<>[]q)</td></tr>
</table>

# Remote references: states

`process_name[process_index]@label`

returns true (1) if the process instance is currently at the local state labeled `label`

`[process_index]` can be omitted if there is a single instance of the process

Can be used in LTL: see `StateLTL.pml`

# Remote references: local variables

`process_name[process_index]@var`

returns value of local variable `var` for a process instance

`[process_index]` can be omitted if there is a single instance of the process

Can be used in LTL