The SPIN Model Checker

Metodi di Verifica del Software

Andrea Corradini – GianLuigi Ferrari

Lezione 4

2011

Slides per gentile concessione di Gerard J. Holzmann

defining correctness properties

- the basic building blocks of a Spin model
  - behavior specification (what is *possible*)
    - asynchronous process behavior
    - variables, data types
    - message channels
  - logical correctness properties (what is *valid*)
    - assertions
    - end-state, progress-state, and acceptance state labels
    - never claims
    - trace assertions
    - temporal logic formulae
    - default properties:
      - absence of system deadlock
      - absence of dead code (unreachable code)

the properties define the real objective of a verification

safety and liveness properties

(a popular terminology due to Leslie Lamport)

safety

- “nothing bad ever happens”
- example: system invariance
  - e.g., ***x*** *is always less than* ***y***
- the model checker’s job is to discover executions that lead to the violation of a safety property (the “bad thing” that should not happen)

liveness

- “something good eventually happens”
- example: responsiveness
  - e.g., *when a request is issued, eventually a response is generated*
- the model checker’s job is to discover executions in which the “good thing” can be postponed indefinitely

syntax for expressing correctness properties

- correctness properties can be expressed:
  - as properties of reachable *states* (safety properties)
  - as properties of *sequences* of states (liveness properties)
- in Promela:

assertions
- local process assertions
- system invariants

end-state labels
- to define proper termination points of processes

properties of states

accept-state labels
- when looking for acceptance *cycles*

progress-state labels
- when looking for *non*-progress *cycles*

never claims (e.g., defined by LTL formulae)

trace assertions

properties of sequences of states

assertions: the oldest type of correctness check

```
byte state = 1;
active proctype A()
{     (state == 1) -> state++;
      assert(state == 2)
}
active proctype B()
{     (state == 1) -> state--;
      assert(state == 0)
}
```

```
$ spin -a simple.pml
$ gcc -o pan pan.c
$ ./pan -E     # -E means ignore invalid endstate errors...
pan: assertion violated (state==2) (at depth 6)
pan: wrote simple.pml.trail
...
```

```
$ spin -t -p simple.pml
  1:    proc  1 (B) line   7 "simple.pml" (state 1) [((state==1))]
  2:    proc  0 (A) line   3 "simple.pml" (state 1) [((state==1))]
  3:    proc  1 (B) line   7 "simple.pml" (state 2) [state--]
  4:    proc  1 (B) line   8 "simple.pml" (state 3) [assert((state==0))]
  5:    proc  0 (A) line   3 "simple.pml" (state 2) [state++]
spin: line   4 "simple.pml", Error: assertion violated
spin: text of failed assertion: assert((state==2))
```

preventing the race

```
byte state = 1;
active proctype A()
{     atomic { (state == 1) -> state++ };
      assert(state == 2)
}
active proctype B()
{     atomic { (state == 1) -> state-- };
      assert(state == 0)
}
```

we added two atomic sequences to create indivisible test&sets

```
$ spin -a simple.pml
$ gcc -o pan pan.c
$ ./pan -E       # -E means ignore invalid endstates...
(Spin Version 4.1.0 -- 6 December 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      - (disabled by -E flag)

State-vector 20 byte, depth reached 3, errors: 0
       6 states, stored
       0 states, matched
       6 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

unreached in proctype A
        (0 of 5 states)
unreached in proctype B
        (0 of 5 states)
```

Q: are there invalid endstates?

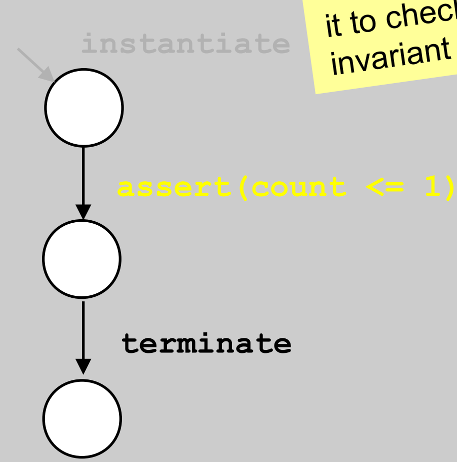
nothing is unreachable

defining system invariants

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
   do
   :: sem!p ->
      sem?v
   od
}

active [5] proctype user()
{
   do
   :: sem?p ->
     count++;
     /* critical section */
     count--;
      sem!v
   od
}
```

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

Q: how expensive is it to check the invariant in this way?

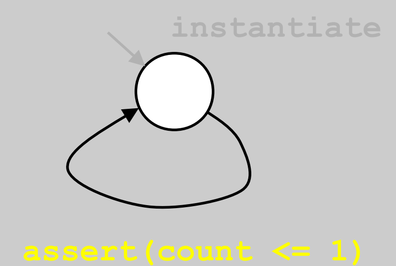


adding active proctype invariant multiplies the search space 3x... (from 16 reachable states to 48)

the more intuitive check

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
   do
   :: sem!p ->
      sem?v
   od
}

active [5] proctype user()
{
   do
   :: sem?p;
     count++;
     /* critical section */
     count--;
      sem!v
   od
}
```

```
active proctype invariant()
{
   do :: assert(count <= 1) od
}
```

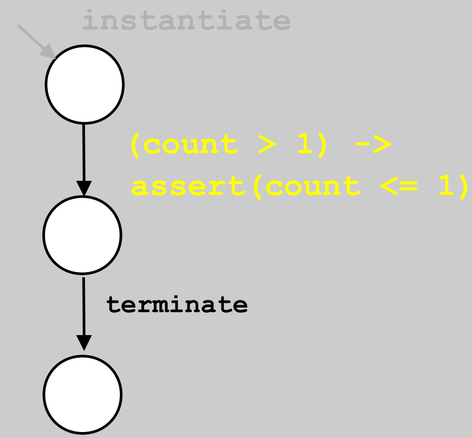


no increase in number of reachable states; but lots of extra transitions explored

better ...

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
   do
   :: sem!p ->
      sem?v
   od
}

active [5] proctype user()
{
   do
   :: sem?p;
     count++;
     /* critical section */
     count--;
      sem!v
   od
}
```

```
active proctype invariant()
{
   d_step { !(count <= 1) ->
            assert(count <= 1) }
}
```

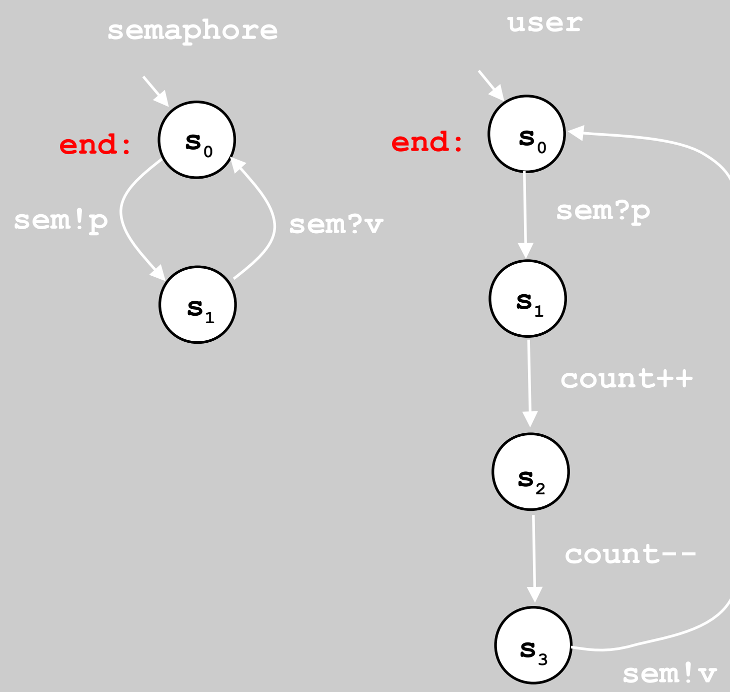


no increase in number of reachable states, no extra transitions

or: put the assertion inside proctype user to check it only when the value of the expression could change

valid end states

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
end:    do
        :: sem!p ->
           sem?v
        od
}

active [5] proctype user()
{
end:    do
        :: sem?p;
      count++;
      /* critical section */
      count--;
           sem!v
        od
}
```

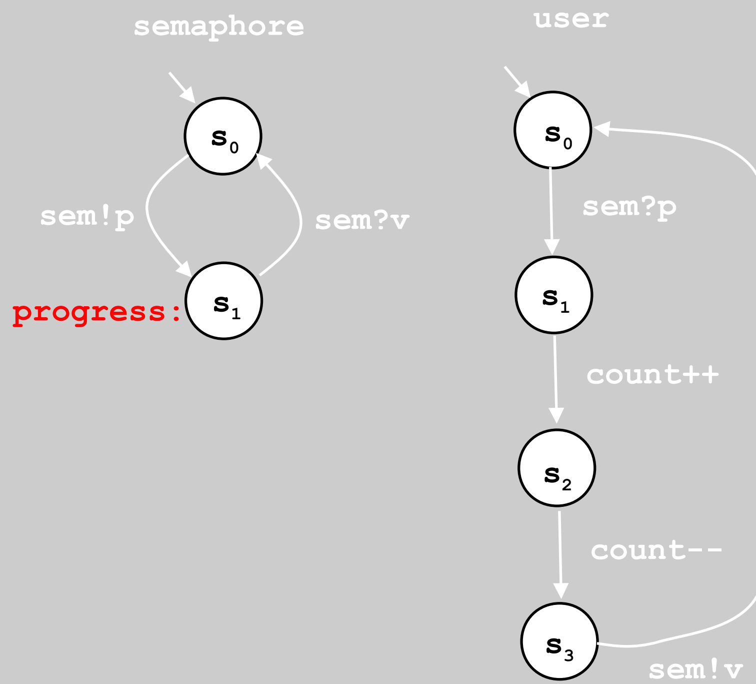


neither process is intended to terminate
the proper endstate in both proctypes is s_0

the model checker can now focus on detecting reachable *invalid* end-states

progress states

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
      do
      :: sem!p ->
progress:   sem?v
      od
}

active [5] proctype user()
{
      do
      :: sem?p ->
       count++;
       /* critical section */
       count--;
            sem!v
      od
}
```



we make effective progress each time a user gains access to the critical section: each time state s_1 is reached in proctype semaphore

the model checker can now focus on detecting reachable *non-progress* cycles

example

```
byte x = 2;

active proctype A()
{
    do
    :: x = 3 - x
    od
}

active proctype B()
{
    do
    :: x = 3 - x
    od
}
```

x alternates between values 2 and 1 ad infinitum
each process has just 1 state
no progress labels used just yet: which by default will mean that every cycle is suspect (i.e., treated as a potential non-progress cycle)

```
$ spin -a fair.pml
$ gcc -DNP -o pan pan.c # non-progress cycle detection
$ ./pan -l              # invoke np-cycle algorithm
pan: non-progress cycle (at depth 2)
pan: wrote fair.pml.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)
State-vector 24 byte, depth reached 7, errors: 1
       3 states, stored (5 visited)
       4 states, matched
       9 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
```

Q1: what happens if we mark *one* of the do-od loops with a progress label?
Q2: what happens if we mark *both* do-od loops?

what kind of cycle did we catch?

that's ok; note that the claim used was predefined during verification with -DNP

```
$ spin -t -p fair.pml
spin: couldn't find claim (ignored)
  2:  proc  1 (B) line  12 "fair.pml" (state 1) [x = (3-x)]
  4:  proc  1 (B) line  12 "fair.pml" (state 1) [x = (3-x)]
  <<<<<START OF CYCLE>>>>>
  6:  proc  1 (B) line  12 "fair.pml" (state 1) [x = (3-x)]
  8:  proc  1 (B) line  12 "fair.pml" (state 1) [x = (3-x)]
spin: trail ends after 8 steps
#processes: 2
                x = 2
  8:  proc  1 (B) line  11 "fair.pml" (state 2)
  8:  proc  0 (A) line   5 "fair.pml" (state 2)
2 processes created
```

we cannot make any assumptions about the relative speeds of processes
it is possible (though not probable) that process B makes infinitely many more steps than process A
the non-progress cycle reported by Spin is not necessarily a *fair* cycle

fair cycles

- we *can* reasonably assume *finite progress*:
  when a process can make progress, it eventually will

there are two commonly used variants:

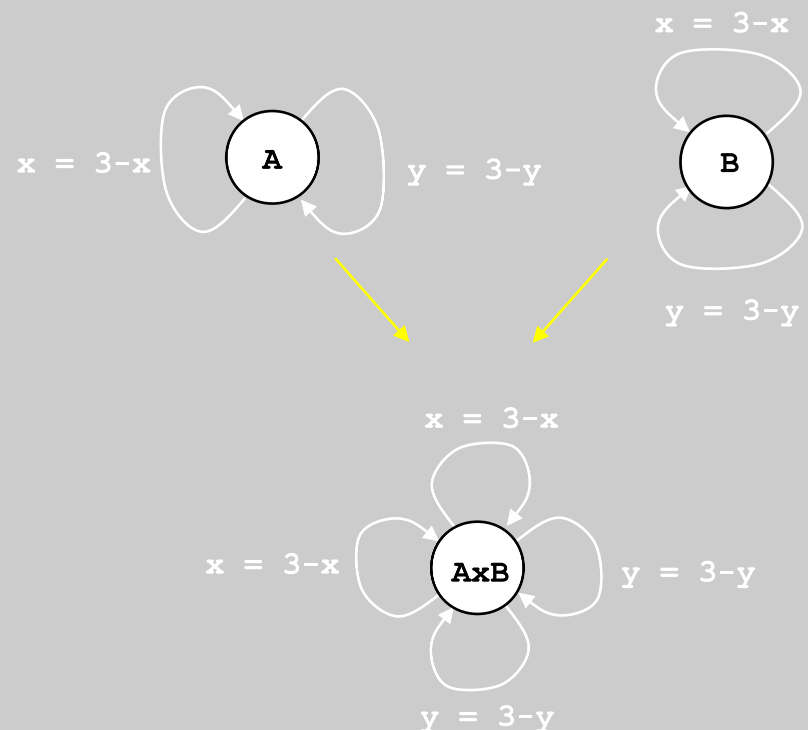
1. *weak* fairness:
   if a statement is executable infinitely *long*,
   it will eventually be executed

2. *strong* fairness:
   if a statement is executable infinitely *often*,
   it will eventually be executed

several interpretations are still possible
fairness can be applied to
1. non-deterministic statement selection *within* a process
2. non-deterministic statement selection *between* processes

statement selection vs process selection

```
byte x = 2, y = 2;

active proctype A()
{
    do
    :: x = 3 - x
    :: y = 3 - y
    od
}

active proctype B()
{
    do
    :: x = 3 - x
    :: y = 3 - y
    od
}
```



Spin contains an algorithm for enforcing *one* case of *weak*-fairness (enabled by run-time option pan -f ...):

> if a *process* contains at least one statement that remains executable infinitely long, that *process* will eventually execute a step

this applies only to potentially *infinite* executions (cycles)

a search for weakly fair non-progress cycles

```
$ ./pan -l -f
pan: non-progress cycle (at depth 8)
pan: wrote fair.pml.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness enabled)
        invalid end states      - (disabled by never claim)
State-vector 24 byte, depth
       4 states, stored (12
       9 states, matched
      21 transitions (= vis
       0 atomic steps
hash conflicts: 0 (resolved
1.573   memory usage (Mbyte
```

```
$ spin -t -p fair.pml
spin: couldn't find claim (ignored)
  2:  proc  1 (B) line  12 "fair.pml" (state 1) [x = (3-x)]
  4:  proc  1 (B) line  12 "fair.pml" (state 1) [x = (3-x)]
  6:  proc  1 (B) line  12 "fair.pml" (state 1) [x = (3-x)]
  8:  proc  0 (A) line   6 "fair.pml" (state 1) [x = (3-x)]
  <<<<<START OF CYCLE>>>>>
 10:  proc  1 (B) line  12 "fair.pml" (state 1) [x = (3-x)]
 12:  proc  1 (B) line  12 "fair.pml" (state 1) [x = (3-x)]
 14:  proc  1 (B) line  12 "fair.pml" (state 1) [x = (3-x)]
 16:  proc  0 (A) line   6 "fair.pml" (state 1) [x = (3-x)]
spin: trail ends after 16 steps
#processes: 2                x = 2
 16:    proc  1 (B) line  11 "fair.pml" (state 2)
 16:    proc  0 (A) line   5 "fair.pml" (state 2)
2 processes created
```

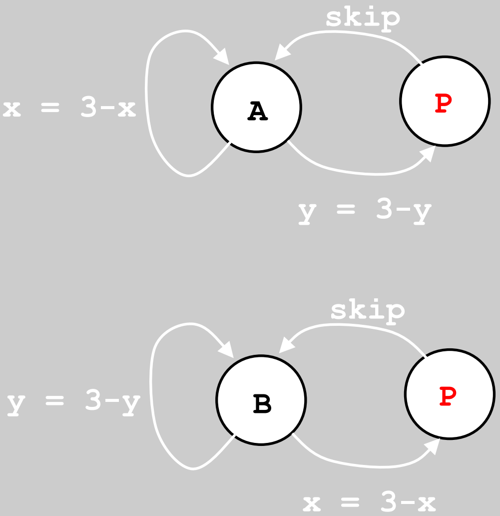
cycle now includes steps from both processes

questions

```
byte x = 2, y = 2;

active proctype A()
{
    do
    :: x = 3 - x
    :: y = 3 - y; progress: skip
    od
}

active proctype B()
{
    do
    :: x = 3 - x; progress: skip
    :: y = 3 - y
    od
}
```

Q1: are there non-progress cycles in this version of the model?

Q2: are there *fair* non-progress cycles in this version of the model?



enforcing fairness constraints

- *any* type of fairness (including the predefined version of weak fairness) can be expressed in LTL formulae
  - we'll return to the use of LTL later

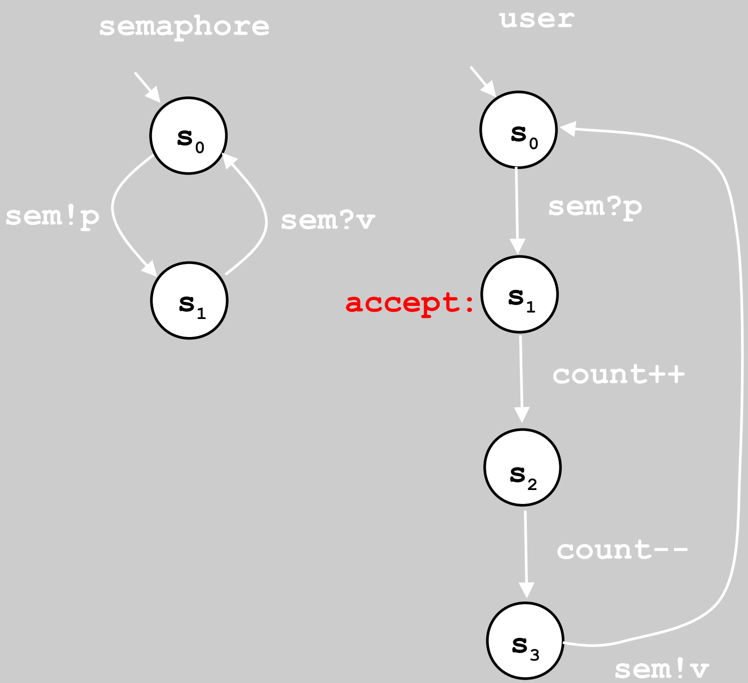
- adding fairness assumptions always increases the cost of verification

- enforcing *strong* fairness constraints is far more costly than enforcing *weak* fairness constraints
  - weak: cost is *linear* in the number of active processes
  - strong: cost is *quadratic* in the number of active processes
  - (cost = increase in time and memory use)

acceptance cycles

marking accept states

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
	do
	:: sem!p ->
		sem?v
	od
}

active [5] proctype user()
{
	do
	:: sem?p ->
accept:	count++;
	/* critical section */
	count--;
		sem!v
	od
}
```



we may want to find infinite executions that *do* pass through a specially marked state

such a state can be identified with an accept-state label

the model checker can now focus on detecting reachable *acceptance* cycles

alternating bit protocol

with lossy transmission and timeout

```
mtype = { msg, ack };
chan to_sndr = [1] of { mtype, bit };
chan to_rcvr = [1] of { mtype, bit };
chan from_sndr = [1] of { mtype, bit };
chan from_rcvr = [1] of { mtype, bit };

active proctype sender()
{   bit a;
    do
    :: from_sndr!msg,a;
       if
       :: to_sndr?ack,eval(a);
          a = 1 - a
       :: timeout /* retransmission */
       fi
    od
}
```

```
active proctype channel()
{   mtype m; bit a;
    do
    :: from_sndr?m,a ->
         if
         :: to_rcvr!m,a
         :: skip /* message loss */
         fi
    :: from_rcvr?m,a ->
         to_sndr!m,a
    od
}
active proctype receiver()
{   bit a;
    do
    :: to_rcvr?msg,eval(a);
         from_rcvr!ack,a;
progress:
         a = 1 - a
    od
}
```

Q1: what constitutes progress?

Q2: is effective progress guaranteed despite the possibility of message loss?

the answer

```
$ spin -a abp2.pml
$ gcc -DNP -o pan pan.c
$ ./pan -l
pan: non-progress cycle (at depth 4)
pan: wrote abp2.pml.trail
...
$
```

```
$ spin -t -c abp2.pml
proc 0 = sender
proc 1 = channel
proc 2 = receiver
spin: couldn't find claim (ignored)
q\p   0   1
  1   from_sndr!msg,0
  1   .   from_sndr?msg,0
  <<<<<START OF CYCLE>>>>>
  1   from_sndr!msg,0
  1   .   from_sndr?msg,0
spin: trail ends after 12 steps
-------------
final state:
-------------
#processes: 3
                queue 1 (from_sndr):
 12: proc  2 (receiver)line 34 "abp2.pml" (state 4)
 12: proc  1 (channel) line 23 "abp2.pml" (state 4)
 12: proc  0 (sender)  line 11 "abp2.pml" (state 5)
3 processes created
$
```

this particular scenario
requires infinitely often
losing the same message

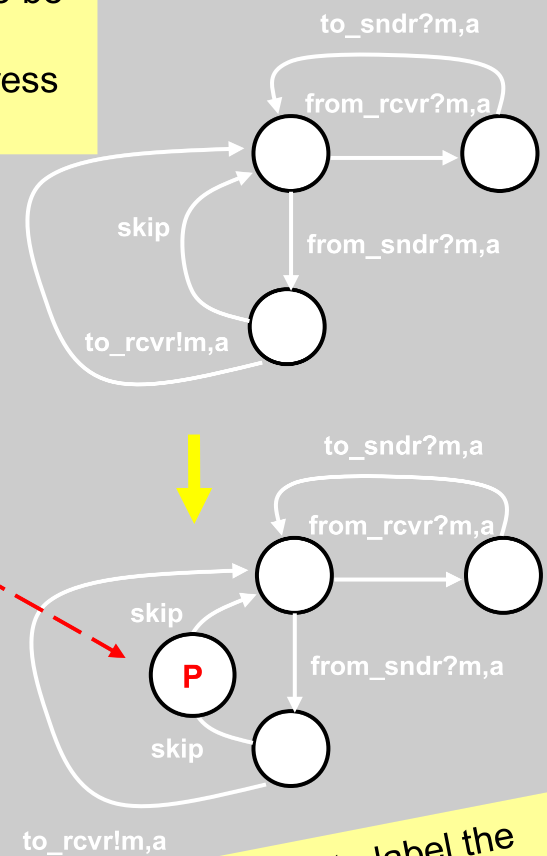
if the probability of loss is <1
then this is an unlikely scenario

Q: can we rule out this scenario
and check for other possible
non-progress cycles?

refining the search

```
active proctype channel()
{   mtype m; bit a;
    do
    :: from_sndr?m,a ->
         if
         :: to_rcvr!m,a
         :: skip; progress: skip /* message loss */
         fi
    :: from_rcvr?m,a ->
         to_sndr!m,a
    od
}
active proctype receiver()
{   bit a;
    do
    :: to_rcvr?msg,eval(a);
         from_rcvr!ack,a;
progress:
         a = 1 - a
    od
}
```

A: consider message loss to be a pseudo ‘progress’ event.... and check if other non-progress cycles are still possible...



be careful to label the right state – if necessary, add a state...

the refined search

```
$ spin -a abp3.pml
$ gcc -DNP -o pan pan.c
$ ./pan -l
(Spin Version 4.1.0 -- 6 December 2003)
        + Partial Order Reduction

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

State-vector 80 byte, depth reached 53, errors: 0
      73 states, stored (98 visited)
      64 states, matched
     162 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

unreached in proctype sender
        line 17, state 10, "-end-"
        (1 of 10 states)
unreached in proctype channel
        line 30, state 12, "-end-"
        (1 of 12 states)
unreached in proctype receiver
        line 40, state 7, "-end-"
        (1 of 7 states)
```

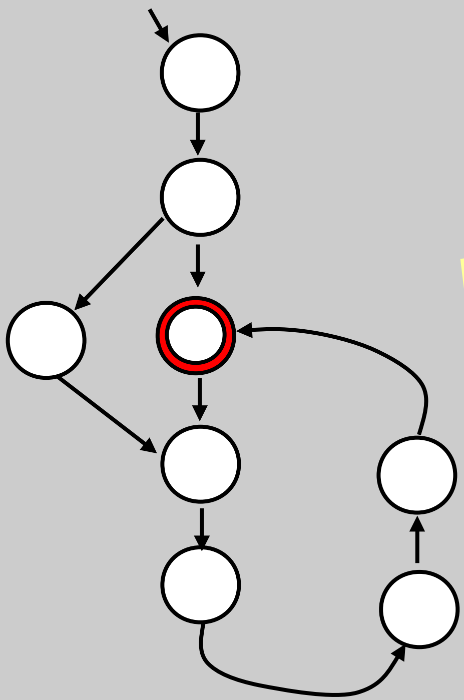
search for non-progress cycles

good news:
no np-cycles remain

meaning: only infinite message loss can cause an infinite delay of progress

why are they called acceptance cycles?

- has to do with the automata theoretic foundation
  - never claims (discussed next) formally define ω-automata that *accept* only those sequences that violate a correctness claim…



acceptance cycle:
a state marked with an accept label that is reachable from the initial system state *and* is also reachable from itself

i.e.,

a strongly connected component in the reachability graph, containing at least one accept state

reviewing

- generic types of properties:
  - assertions
    - local process assertions
    - system invariants
  - end-state labels
    - to define proper termination points of processes
  - accept-state labels
    - when looking for acceptance *cycles*
  - progress-state labels
    - when looking for *non*-progress *cycles*
  - never claims (optionally derived from LTL formulae)
  - trace assertions

states

cycles

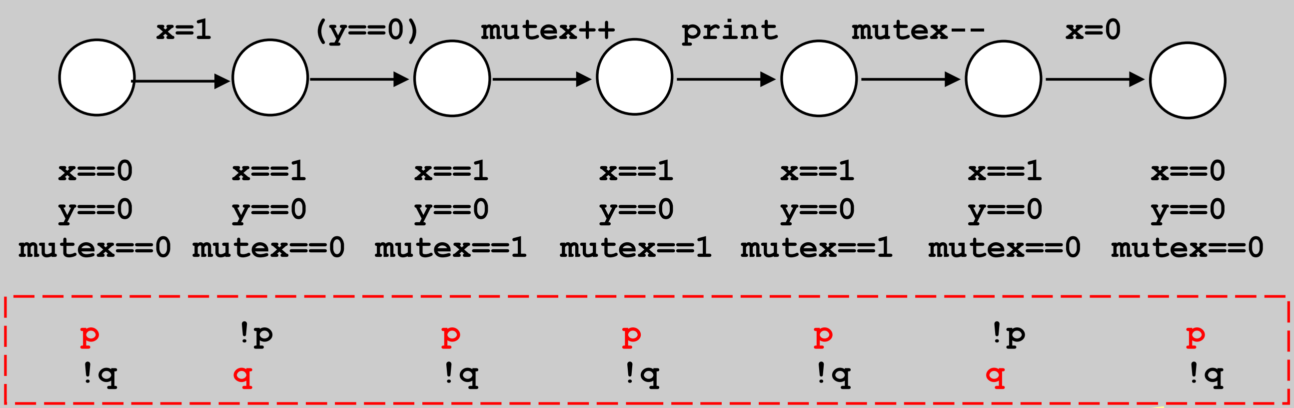
combinations of accept and progress labels with or without the weak fairness constraint can already express a range of different liveness properties

reasoning about executions

- there are at least three different ways to formalize an execution in a concurrent system:
  - sequence of states
  - sequence of events (state transitions)
  - sequence of propositions on states (state properties)

this is what Spin does

```
bit  x, y;
byte mutex;
active proctype A() {
      x = 1;
     (y == 0) ->
      mutex++;
      printf("%d\n", _pid);
      mutex--;
      x = 0
}
```



properties of states

p: (x == mutex)
q: (x != y)

is it always true that p *implies* !q ?

reasoning about executions

- checking for every state that (p *implies* !q) is simple – it is a system invariant that we can check with a monitor process:

```
active proctype invariant() {
   do
   :: assert(!p || !q)   /* p implies !q */
   od
}
```

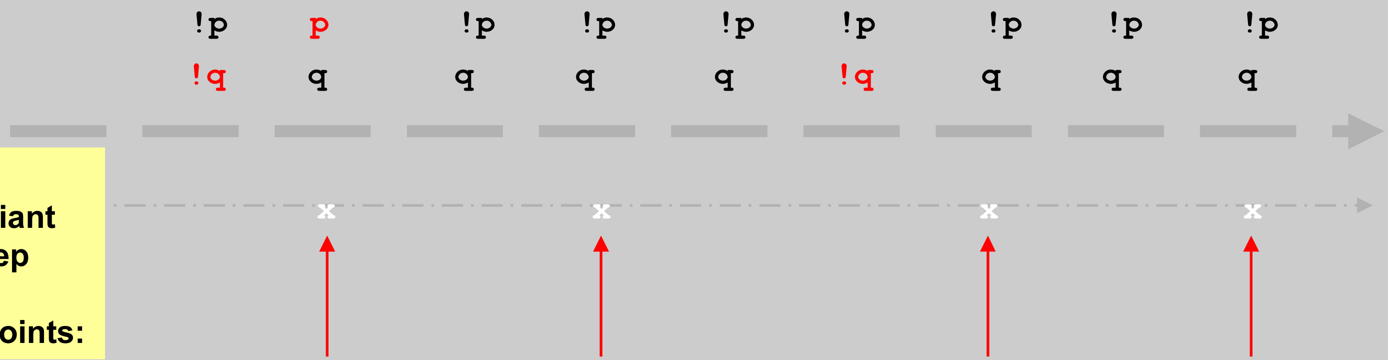
- but now consider checking:
  - every state where property ***p*** holds is *followed* by a state where property !***q*** holds (a *temporal* instead of a *causal* property)
  - this does not work:

```
active proctype invariant()   /* first try */
{
   (p) ->             /* after p holds */
accept:
   do
   :: (q)             /* then forever q is bad */
   od
}
```

wrong

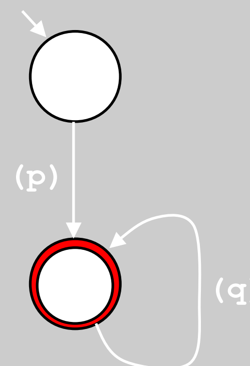
why it does not work

consider this
execution



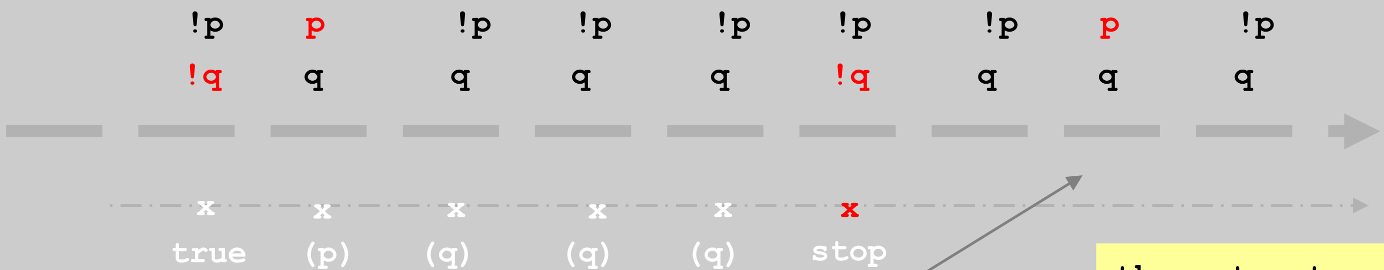
assume
process invariant
executes a step
only at these
interleaving points:

```
active proctype invariant() {
    (p) ->
accept:
    do
    :: (q) /* first p and then forever q is bad */
    od
}
```



we cannot assume *anything*
about the relative speed of
execution of any process...

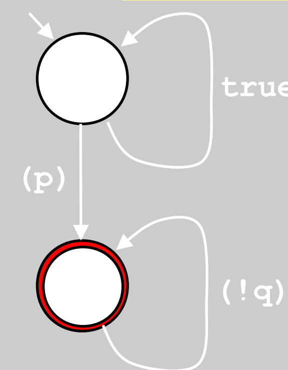
the checker for a property of this type must execute *synchronously* with the system




```
never {
    do
    :: true
    :: (p) -> break
    od;
accept:
    do
    :: (q) /* first p and then forever q is bad */
    od
}
```

be prepared to wait for p to become true at *any* point in the execution

the automaton can be non-deterministic



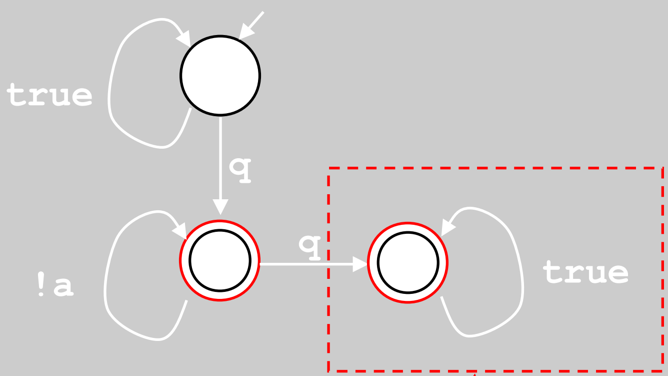
a never claim executes an expression statement at every step in an execution

never claims are intended to observe system behavior they should *not* contribute to system behavior

the never claim tracks behavior and can identify the *bad* executions (in this case with an accept label)

a different property

- question ***q*** is always eventually followed by answer ***a*** (assume ***q*** and ***a*** are properties of states) *BEFORE the next question is asked...*
- this requirement is *violated* by any execution where a ***q*** is not followed by an ***a*** at all, AND by any execution where a ***q*** follows a ***q*** without an ***a*** in between




```
never {
      do
      :: true
      :: q -> break
      od;
accept0:  do
      :: !a
      :: q -> break
      od;
accept1:  do
      :: true
      od
}
```

reaching the end of a never claim is an automatic error

we *can* (but need not) make this explicit; as is done here

conventions

```
never {
      do
      :: true
      :: q -> break
      od;
accept0:  do
      :: !a
      :: q -> break
      od;
accept1:  do
      :: true
      od
}
```

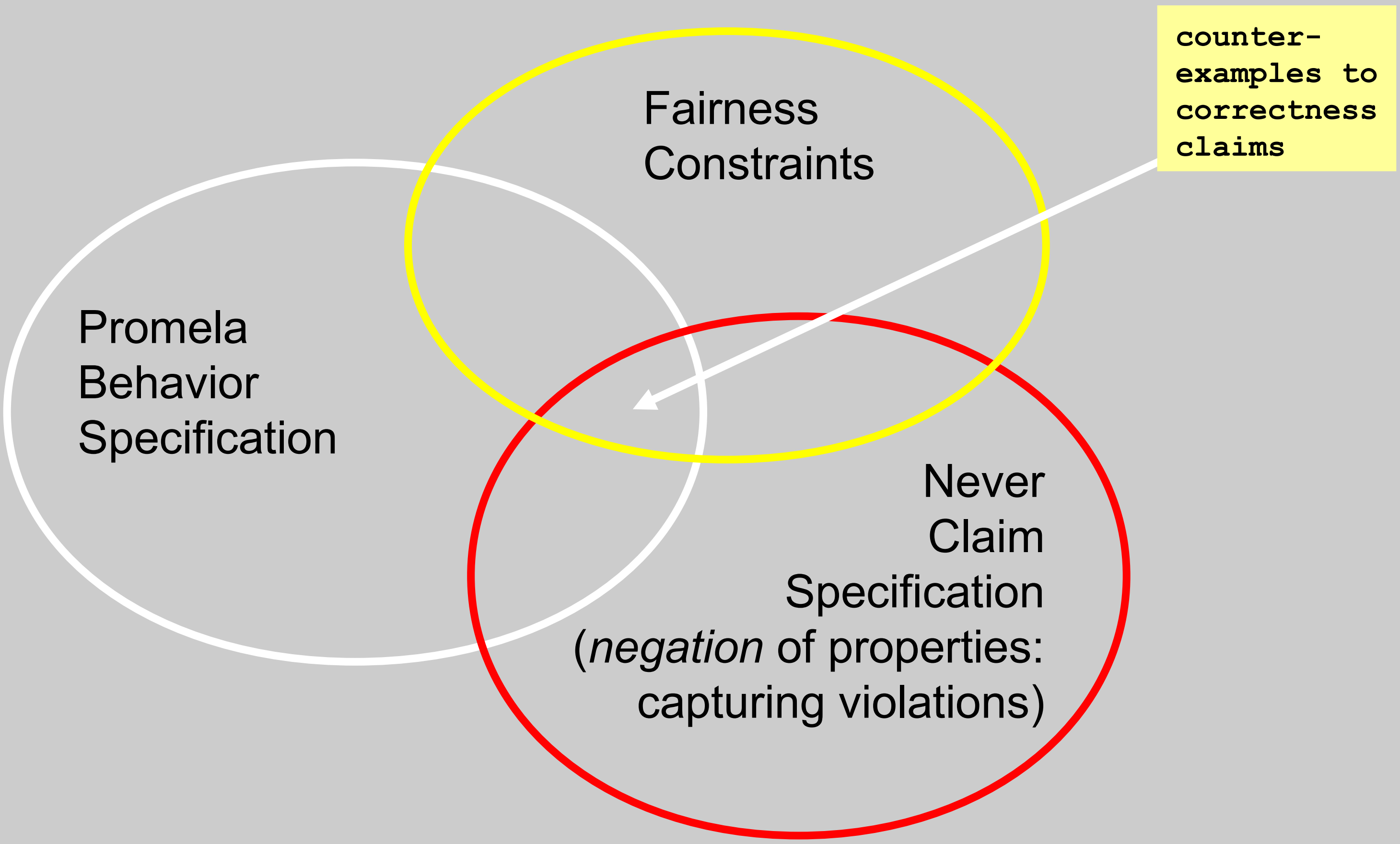
↔

```
never {
      do
      :: true
      :: q -> break
      od;
accept0:  do
      :: !a
      :: q -> break
      od
}
```

reaching the closing curly brace of a never claim means that the entire behavior pattern that was expressed was *matched*, and is always interpreted as an error (it should ***never*** happen)

never claims are designed to '***accept***' bad behavior – property ***violations***

the language intersection picture



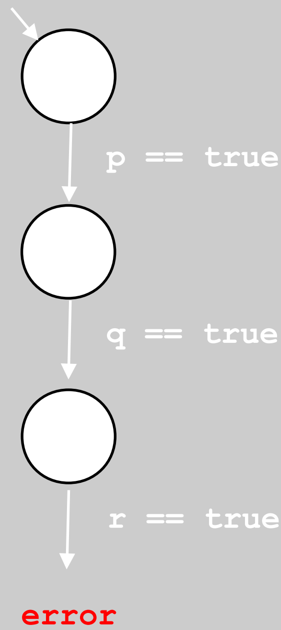
a longer temporal sequence

- there is no execution where first p becomes true, then q, and then r

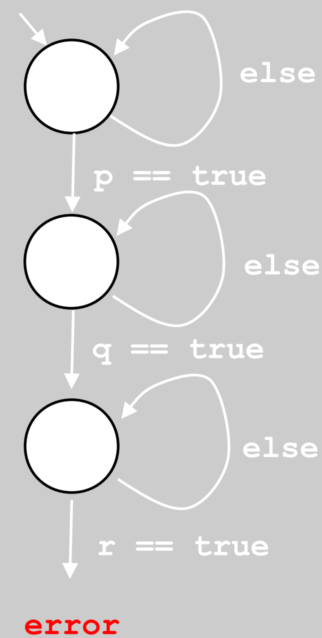
```
/* first try: */

never {
   p; q; r
}
```

incorrect

monitors only the first 3 steps in any execution....




```
never {
   do
   :: p -> break
   :: else
   od;
   do
   :: q -> break
   :: else
   od;
   do
   :: r -> break
   :: else
   od
}
```



correct version

applies to an execution of any length