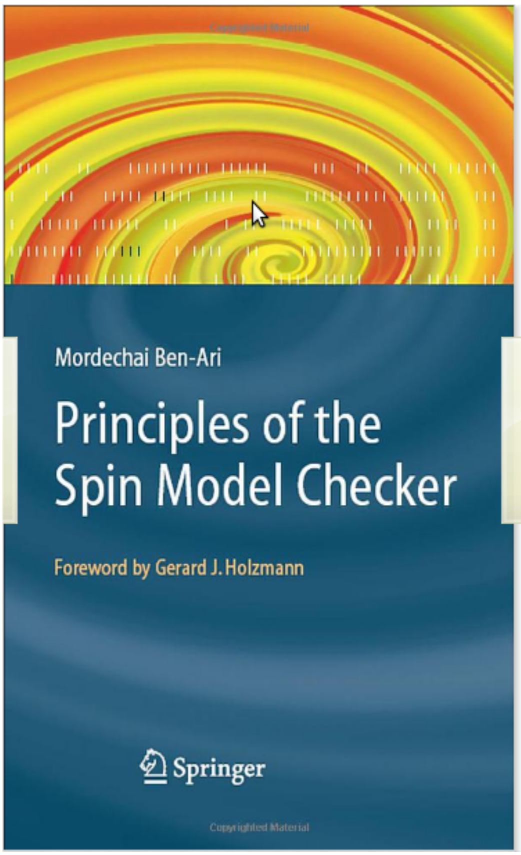
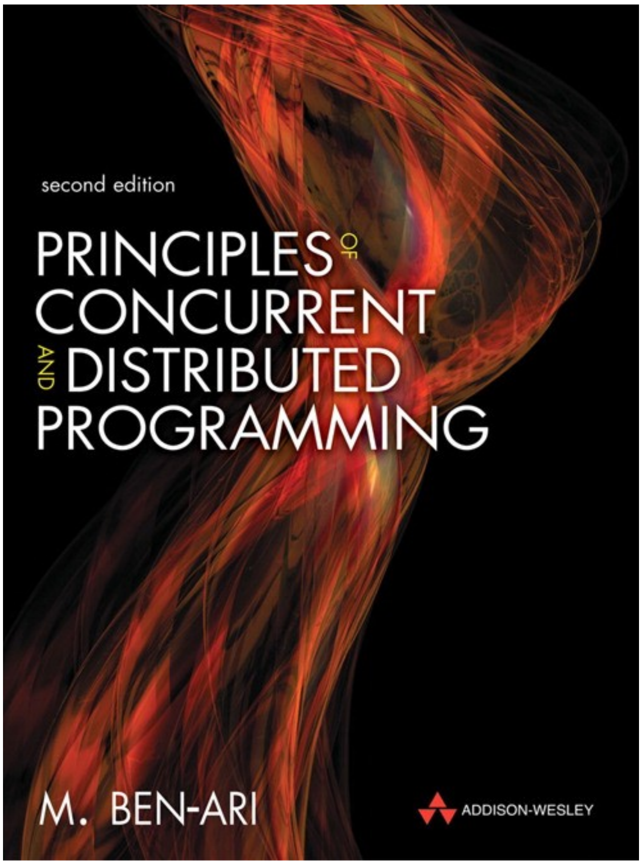


Mordechai Ben-Ari

Principles of the Spin Model Checker

Springer, 2008

ISBN: 978-1-84628-769-5



Mordechai Ben-Ari

Principles of Concurrent and Distributed Programming

Second Edition
Addison-Wesley, 2008

ISBN: 978-0-32131-283-9

http://www.springer.com/computer/swe/book/978-1-84628-769-5

Supplementary material (zip, 38 kB)

978-1-84628-769-5-additional material.zip

Chapter 5

Verification with Temporal Logic

Section 5.4

Liveness properties

Let A be a LTL formula and let $\tau = (s_0, s_1, s_2, ...)$ be a computation. Then $\diamond A$, read *diamond* or *eventually* A, is true in state s_i if and only if A is true *for some* s_j in τ such that $j \geq i$.

The operator is reflexive, so if A is true in state s, then so is $\diamond A$.

The formula $\diamond A$ is called a *liveness property* because it specifies that something "good" eventually happens in the computation.

If *csp* is the atomic proposition that is true in a state if process P is in its critical section, then $\Diamond$*csp* holds if and only if process P eventually enters its critical section.

It is essential that correctness specifications contain liveness properties because a safety property is vacuously satisfied by an empty program that does nothing! For example, a solution to the critical section problem in which neither process tries to enter its critical section trivially fulfills the correctness properties of mutual exclusion and absence of deadlock:

```
start:
  do
  :: printf("Noncritical section\n")
     goto start
     wantP = true  /* Try to enter the critical section */
     printf("Critical section\n")
  od
```

Subsection 5.4.1

Expressing liveness properties in Spin

Listing 5.1. Critical section with starvation (4th attempt)

```
 1  bool wantP = false, wantQ = false
 2
 3  active proctype P() {
 4    do
 5    :: wantP = true
 6      do
 7      :: wantQ ->
 8           wantP = false
 9           wantP = true
10      :: else -> break
11      od
12      wantP = false
13    od
17  }
15
16  active proctype Q() {
17    do
18    :: wantQ = true
19      do
20      :: wantP ->
21           wantQ = false
22           wantQ = true
23      :: else -> break
24      od
25      wantQ = false
26    od
27  }
```

Unfortunately, this program is not fully correct because starvation may occur, that is, there is a computation in which process P never enters its critical section:

s_0 = (5. wantP=1, 18. wantQ=1, 0, 0) →

s_1 = (5. wantP=1, 20. wantP, 0, 1) →

s_2 = (5. wantP=1, 25. wantQ=0, 0, 1) →

s_3 = (7. wantQ, 25. wantQ=0, 1, 1) →

s_4 = (8. wantP=0, 25. wantQ=0, 1, 1) →

s_5 = (9. wantP=1, 25. wantQ=0, 0, 1) →

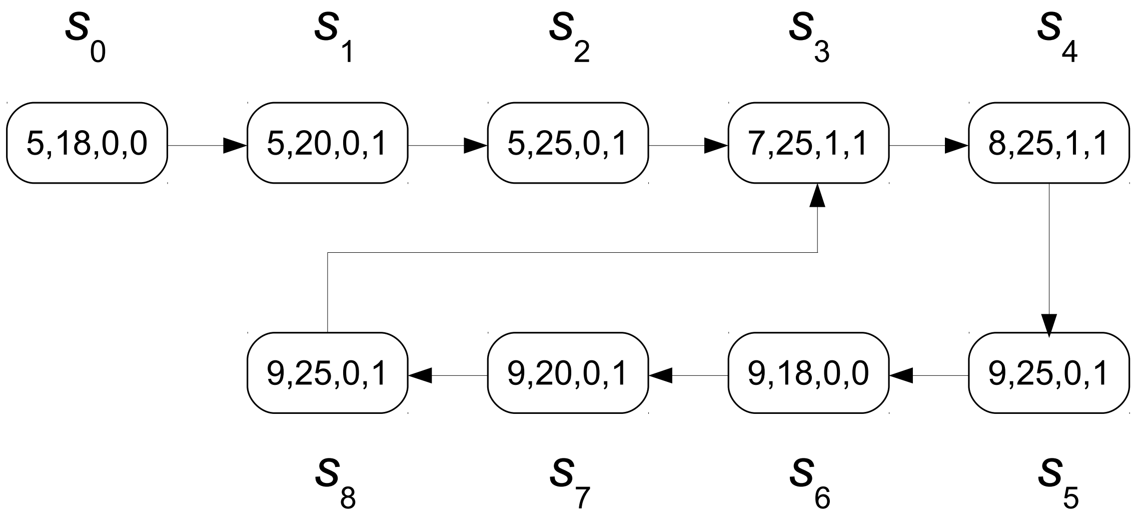
s_6 = (9. wantP=1, 18. wantQ=1, 0, 0) →

s_7 = (9. wantP=1, 20. wantP, 0, 1) →

s_8 = (9. wantP=1, 25. wantQ=0, 0, 1) →

s_9 = (7. wantQ, 25. wantQ=0, 1, 1)

Since state s_9 is the same as state s_3, they can be identified and the sequence of states extended to an infinite computation:



The critical section of process P (line 12) does not appear in any state of this computation, demonstrating that absence of starvation does not hold for this program.

Subsection 5.4.2

Verifying liveness properties in Spin

Add the statements

```
        csp = true
        csp = false
```

between lines 11 and 12 of the program in Listing 5.1;

then LTL formula `<>csp` expresses absence of starvation for process P.

The verification of the temporal formula is carried out in a manner similar to that of the safety property, except that it must be performed in a mode called searching for *acceptance cycles* (Section 10.3.2).

Weak fairness, explained in Section 5.5, must also be specified when this program is verified.

File `fourth-liveness_.pml`

```
 1  /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */
 2
 3  bool wantP = false, wantQ = false
 4  bool csp = false
 5
 6  active proctype P() {
 7    do
 8    :: wantP = true
 9       do
10       :: wantQ -> wantP = false; wantP = true
11       :: else  -> break
12       od
13       csp = true
14       csp = false
15       wantP = false
16    od
17  }
18
19  active proctype Q() {
20    do
21    :: wantQ = true
22       do
23       :: wantP -> wantQ = false; wantQ = true
24       :: else -> break
25       od
26       wantQ = false
27    od
28  }
```

```
$                      # -a (acceptance), -f (weak fairness)
$ spin -f '!<>csp' -run -a -f forth-liveness_.pml
warning: for p.o. reduction to be valid the never claim must be stutter-
invariant
(never claims generated from LTL formulae are stutter-invariant)
pan:1: acceptance cycle (at depth 14)
pan: wrote fourth-liveness_.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             + (never_0)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness enabled)
        invalid end states      - (disabled by never claim)

State-vector 36 byte, depth reached 51, errors: 1
       26 states, stored (52 visited)
       18 states, matched
       70 transitions (= visited+matched)
        0 atomic steps
...
```

File fourth-liveness2.pml

```
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

bool wantP = false, wantQ = false
bool csp = false

ltl p1 { <>csp }

active proctype P() {
  do
  :: wantP = true
     do
     :: wantQ -> wantP = false; wantP = true
     :: else  -> break
     od
     csp = true
     csp = false
     wantP = false
  od
}

active proctype Q() {
  do
  :: wantQ = true
     do
     :: wantP -> wantQ = false; wantQ = true
     :: else -> break
     od
     wantQ = false
  od
}
```

```
$         # -a (acceptance), -f (weak fairness)
$ spin -run -a -f forth-liveness2.pml
ltl p1: <> (csp)
pan:1: acceptance cycle (at depth 14)
pan: wrote fourth-liveness2.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             + (p1)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness enabled)
        invalid end states      - (disabled by never claim)

State-vector 36 byte, depth reached 51, errors: 1
       26 states, stored (52 visited)
       18 states, matched
       70 transitions (= visited+matched)
        0 atomic steps
...
```

Liveness does not hold for this program; the error message is

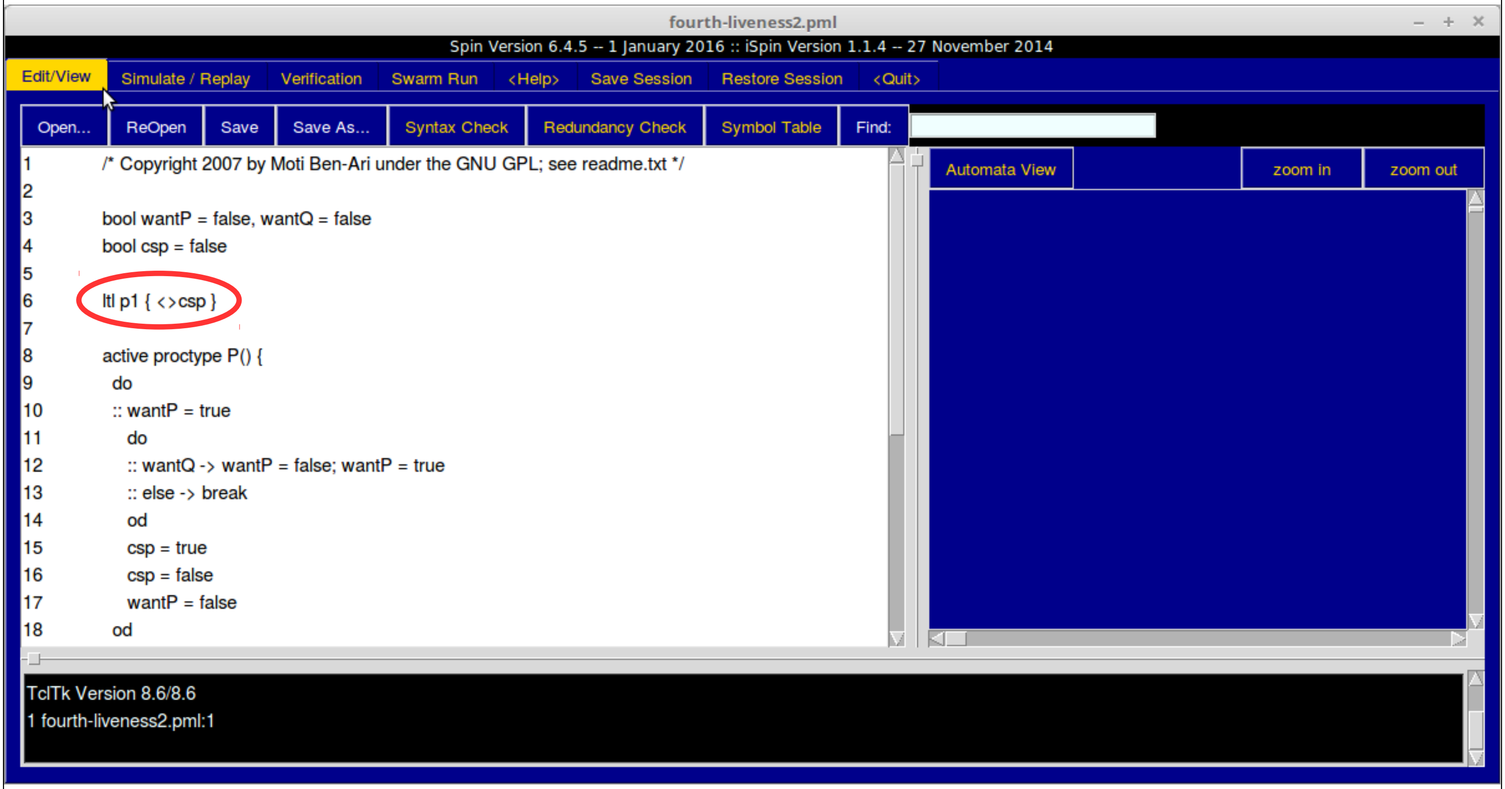
```
pan:1: acceptance cycle (at depth 14)
pan: wrote fourth-liveness2.pml.trail
```

For safety properties, a counterexample consists of one state where the formula is false,

but for a liveness property, a counterexample is an infinite computation in which something good – in this case, `csp` becomes true – never happens. To produce the counterexample, run a guided simulation with the trail.

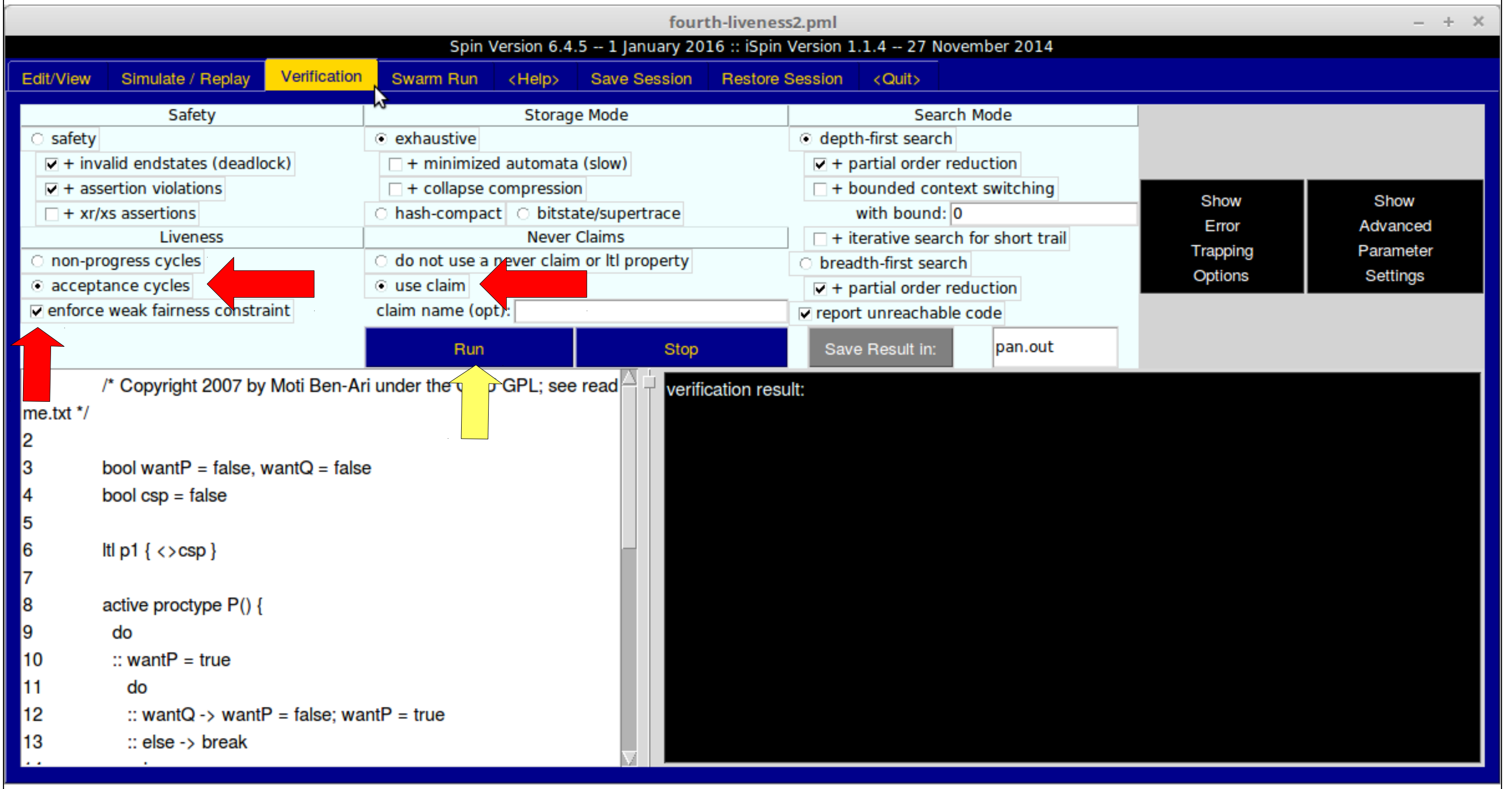
```
$ spin -t -p -g forth-liveness2.pml
ltl p1: <> (csp)
starting claim 2
using statement merging
Never claim moves to line 4     [(!(csp))]
  2:    proc  1 (Q:1) fourth-liveness2.pml:23 (state 1) [wantQ = 1]
                wantQ = 1
  4:    proc  1 (Q:1) fourth-liveness2.pml:26 (state 5) [else]
  6:    proc  1 (Q:1) fourth-liveness2.pml:28 (state 10)        [wantQ = 0]
                wantQ = 0
  8:    proc  0 (P:1) fourth-liveness2.pml:10 (state 1) [wantP = 1]
                wantP = 1
 10:    proc  1 (Q:1) fourth-liveness2.pml:23 (state 1) [wantQ = 1]
                wantQ = 1
 12:    proc  1 (Q:1) fourth-liveness2.pml:25 (state 2) [(wantP)]
 14:    proc  0 (P:1) fourth-liveness2.pml:12 (state 2) [(wantQ)]
  <<<<<START OF CYCLE>>>>>
 16:    proc  1 (Q:1) fourth-liveness2.pml:25 (state 3) [wantQ = 0]
                wantQ = 0
 18:    proc  1 (Q:1) fourth-liveness2.pml:25 (state 4) [wantQ = 1]
                wantQ = 1
 20:    proc  1 (Q:1) fourth-liveness2.pml:25 (state 2) [(wantP)]
 22:    proc  0 (P:1) fourth-liveness2.pml:12 (state 3) [wantP = 0]
                wantP = 0
 24:    proc  1 (Q:1) fourth-liveness2.pml:25 (state 3) [wantQ = 0]
                wantQ = 0
 26:    proc  1 (Q:1) fourth-liveness2.pml:25 (state 4) [wantQ = 1]
                wantQ = 1
...
```

```
...
 28:    proc  1 (Q:1) fourth-liveness2.pml:26 (state 5) [else]
 30:    proc  0 (P:1) fourth-liveness2.pml:12 (state 4) [wantP = 1]
                wantP = 1
 32:    proc  0 (P:1) fourth-liveness2.pml:12 (state 2) [(wantQ)]
 34:    proc  1 (Q:1) fourth-liveness2.pml:28 (state 10)        [wantQ = 0]
                wantQ = 0
 36:    proc  0 (P:1) fourth-liveness2.pml:12 (state 3) [wantP = 0]
                wantP = 0
 38:    proc  1 (Q:1) fourth-liveness2.pml:23 (state 1) [wantQ = 1]
                wantQ = 1
 40:    proc  1 (Q:1) fourth-liveness2.pml:26 (state 5) [else]
 42:    proc  1 (Q:1) fourth-liveness2.pml:28 (state 10)        [wantQ = 0]
                wantQ = 0
 44:    proc  0 (P:1) fourth-liveness2.pml:12 (state 4) [wantP = 1]
                wantP = 1
 46:    proc  1 (Q:1) fourth-liveness2.pml:23 (state 1) [wantQ = 1]
                wantQ = 1
 48:    proc  1 (Q:1) fourth-liveness2.pml:25 (state 2) [(wantP)]
 50:    proc  0 (P:1) fourth-liveness2.pml:12 (state 2) [(wantQ)]
spin: trail ends after 50 steps
#processes: 2
                wantP = 1
                wantQ = 1
                csp = 0
 50:    proc  1 (Q:1) fourth-liveness2.pml:25 (state 3)
 50:    proc  0 (P:1) fourth-liveness2.pml:12 (state 3)
 50:    proc  - (p1:1) _spin_nvr.tmp:3 (state 3)
2 processes created
```



```
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

bool wantP = false, wantQ = false
bool csp = false

ltl p1 { <>csp }

active proctype P() {
  do
  :: wantP = true
     do
     :: wantQ -> wantP = false; wantP = true
     :: else -> break
     od
     csp = true
     csp = false
     wantP = false
  od
```

```
TclTk Version 8.6/8.6
1 fourth-liveness2.pml:1
```


fourth-liveness2.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Run
Stop
Show Error Trapping Options
Show Advanced Parameter Settings

Safety
- ○ safety
- ☑ + invalid endstates (deadlock)
- ☑ + assertion violations
- □ + xr/xs assertions

Liveness
- ○ non-progress cycles
- ● acceptance cycles
- ☑ enforce weak fairness constraint

Storage Mode
- ● exhaustive
- □ + minimized automata (slow)
- □ + collapse compression
- ○ hash-compact ○ bitstate/supertrace

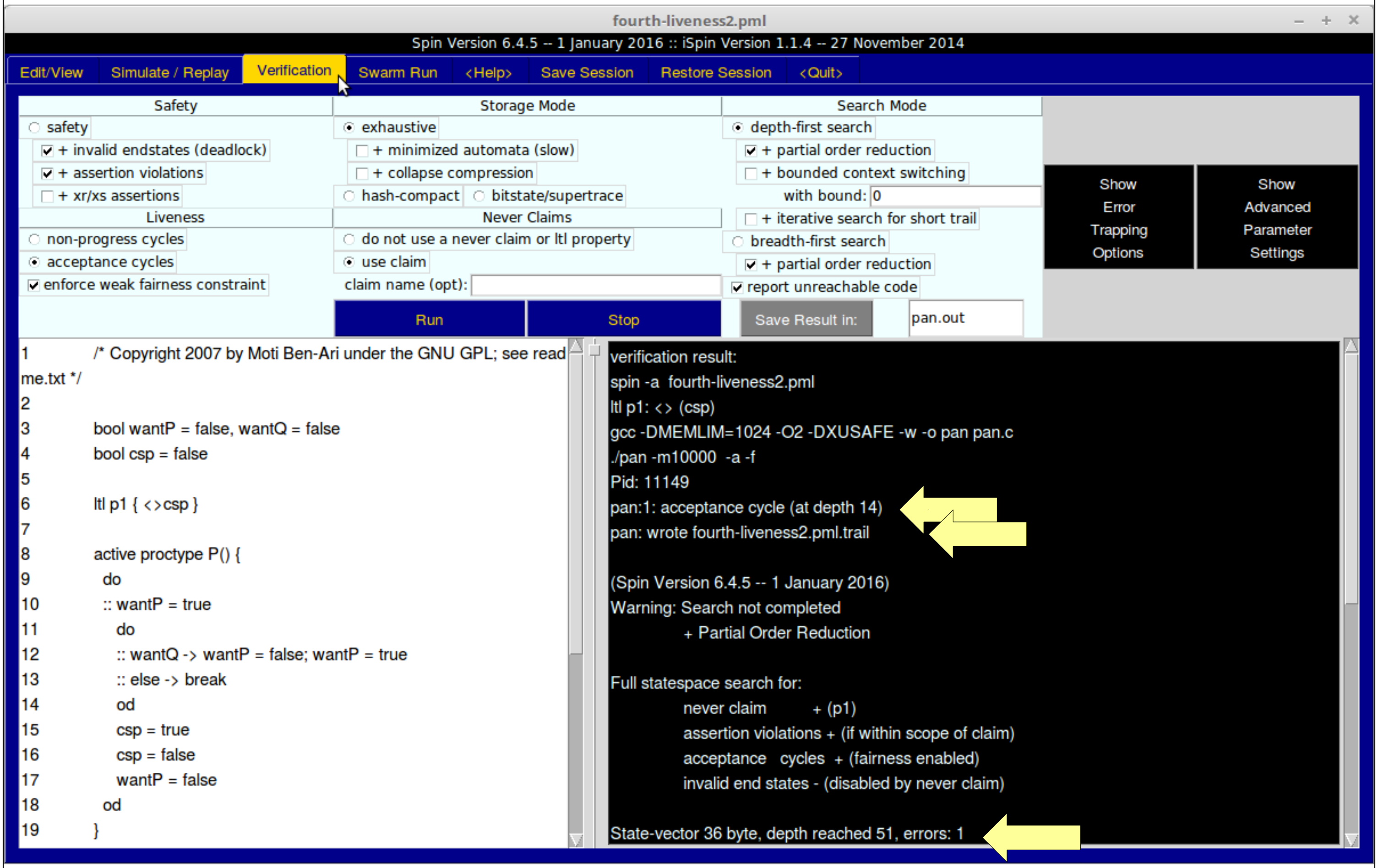
Never Claims
- ○ do not use a never claim or ltl property
- ● use claim
- claim name (opt):

Search Mode
- ● depth-first search
- ☑ + partial order reduction
- □ + bounded context switching
- with bound: 0
- □ + iterative search for short trail
- ○ breadth-first search
- ☑ + partial order reduction
- ☑ report unreachable code

Save Result in: pan.out

```
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see read
me.txt */

bool wantP = false, wantQ = false
bool csp = false

ltl p1 { <>csp }

active proctype P() {
  do
  :: wantP = true
     do
     :: wantQ -> wantP = false; wantP = true
     :: else -> break
```

```
verification result:
```



fourth-liveness2.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Run
Stop
Show Error Trapping Options
Show Advanced Parameter Settings

Safety
○ safety
☑ + invalid endstates (deadlock)
☑ + assertion violations
□ + xr/xs assertions

Liveness
○ non-progress cycles
● acceptance cycles
☑ enforce weak fairness constraint

Storage Mode
● exhaustive
□ + minimized automata (slow)
□ + collapse compression
○ hash-compact ○ bitstate/supertrace

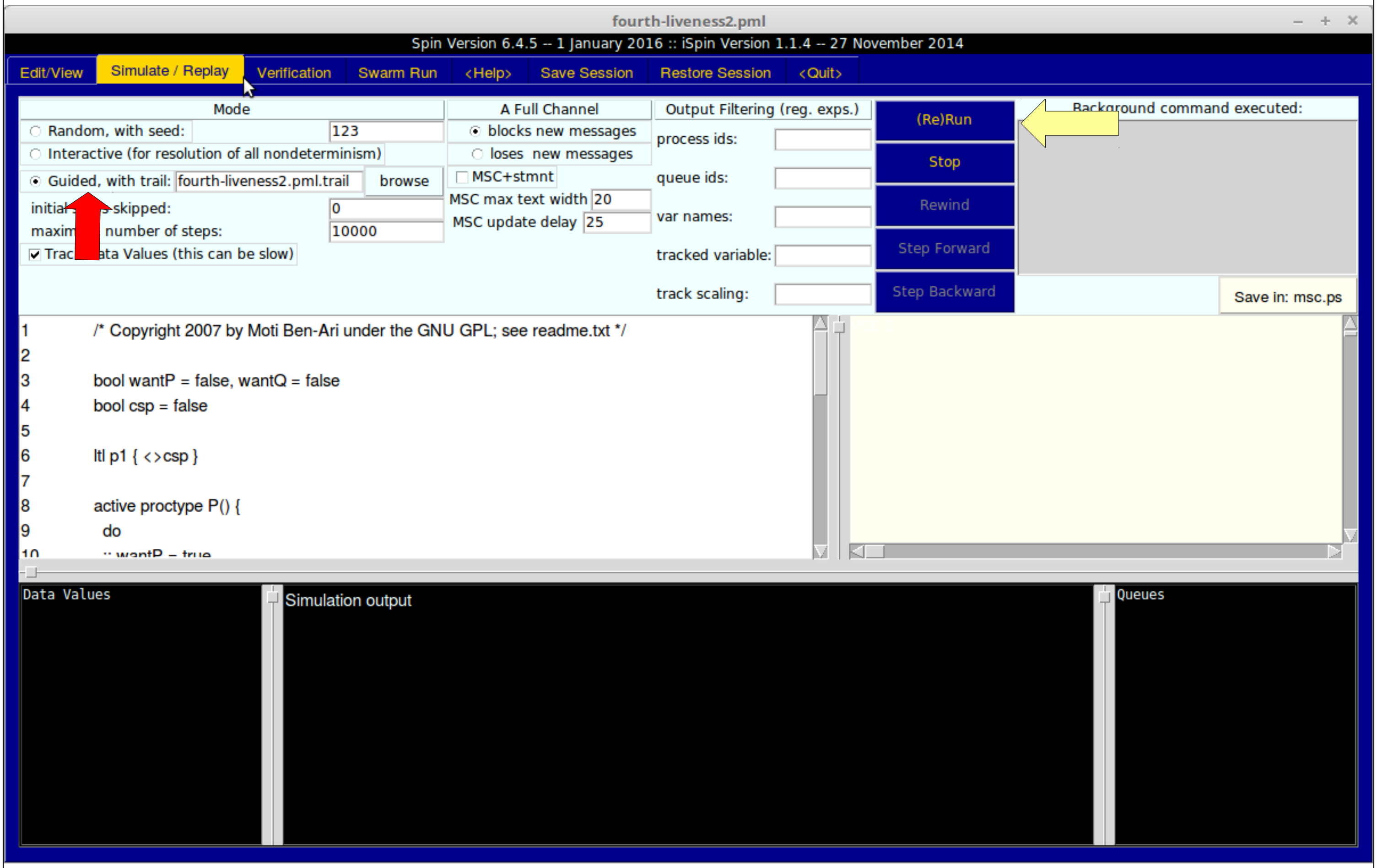
Never Claims
○ do not use a never claim or ltl property
● use claim
claim name (opt):

Search Mode
● depth-first search
☑ + partial order reduction
□ + bounded context switching
with bound: 0
□ + iterative search for short trail
○ breadth-first search
☑ + partial order reduction
☑ report unreachable code

Save Result in: pan.out

```
    /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

    bool wantP = false, wantQ = false
    bool csp = false

    ltl p1 { <>csp }

    active proctype P() {
      do
      :: wantP = true
         do
         :: wantQ -> wantP = false; wantP = true
         :: else -> break
         od
         csp = true
         csp = false
         wantP = false
      od
    }
```

```
verification result:
spin -a  fourth-liveness2.pml
ltl p1: <> (csp)
gcc -DMEMLIM=1024 -O2 -DXUSAFE -w -o pan pan.c
./pan -m10000  -a -f
Pid: 11149
pan:1: acceptance cycle (at depth 14)
pan: wrote fourth-liveness2.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim         + (p1)
        assertion violations + (if within scope of claim)
        acceptance   cycles  + (fairness enabled)
        invalid end states - (disabled by never claim)

State-vector 36 byte, depth reached 51, errors: 1
```



Mode
○ Random, with seed: 123
○ Interactive (for resolution of all nondeterminism)
● Guided, with trail: fourth-liveness2.pml.trail
initial steps skipped: 0
maximum number of steps: 10000
☑ Track Data Values (this can be slow)

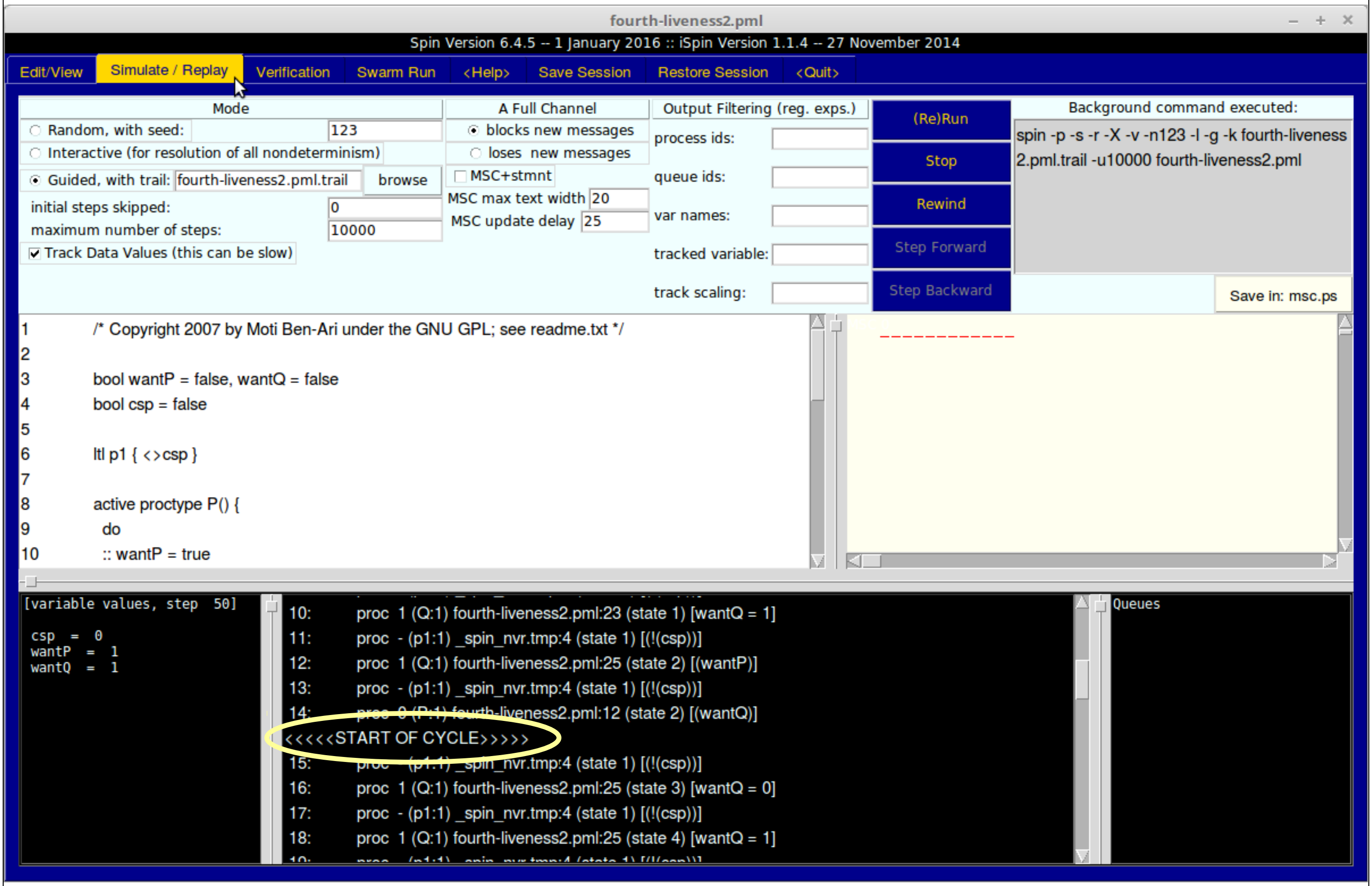
A Full Channel
● blocks new messages
○ loses new messages
□ MSC+stmnt
MSC max text width 20
MSC update delay 25

Output Filtering (reg. exps.)
process ids:
queue ids:
var names:
tracked variable:
track scaling:

Background command executed:

Save in: msc.ps

```
    /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

    bool wantP = false, wantQ = false
    bool csp = false

    ltl p1 { <>csp }

    active proctype P() {
      do
      :: wantP = true
```



○ Random, with seed: 123
○ Interactive (for resolution of all nondeterminism)
● Guided, with trail: fourth-liveness2.pml.trail
initial steps skipped: 0
maximum number of steps: 10000
☑ Track Data Values (this can be slow)

● blocks new messages
○ loses new messages
□ MSC+stmnt
MSC max text width 20
MSC update delay 25

process ids:
queue ids:
var names:
tracked variable:
track scaling:

```
spin -p -s -r -X -v -n123 -l -g -k fourth-liveness2.pml.trail -u10000 fourth-liveness2.pml
```

```
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

bool wantP = false, wantQ = false
bool csp = false

ltl p1 { <>csp }

active proctype P() {
  do
  :: wantP = true
```

```
[variable values, step  50]

 csp  =  0
 wantP  =  1
 wantQ  =  1
```

```
10:     proc  1 (Q:1) fourth-liveness2.pml:23 (state 1) [wantQ = 1]
11:     proc  - (p1:1) _spin_nvr.tmp:4 (state 1) [(!(csp))]
12:     proc  1 (Q:1) fourth-liveness2.pml:25 (state 2) [(wantP)]
13:     proc  - (p1:1) _spin_nvr.tmp:4 (state 1) [(!(csp))]
14:     proc  0 (P:1) fourth-liveness2.pml:12 (state 2) [(wantQ)]
<<<<<START OF CYCLE>>>>>
15:     proc  - (p1:1) _spin_nvr.tmp:4 (state 1) [(!(csp))]
16:     proc  1 (Q:1) fourth-liveness2.pml:25 (state 3) [wantQ = 0]
17:     proc  - (p1:1) _spin_nvr.tmp:4 (state 1) [(!(csp))]
18:     proc  1 (Q:1) fourth-liveness2.pml:25 (state 4) [wantQ = 1]
19:     proc  - (p1:1) _spin_nvr.tmp:4 (state 1) [(!(csp))]
```

The line **START OF CYCLE** indicates that the subsequent states form a cycle that can be repeated indefinitely. Since a variable appears in the Spin output only when it is assigned to, the absence of a value for `csp` means that the variable has never been assigned to and hence that starvation occurs in this computation.

Advanced: Finding the shortest counterexamples

Spin did not find the *shortest* counterexample. That is because Spin performs a depth-first search of the state diagram and stops with the first counterexample it finds. The `-i` and `-I` arguments to `pan` can be used to perform an iterated search for shorter counterexamples; see pages 24-25 of *Spin Model Checker* for details.

Section 5.5

Fairness

Consider again the program for the critical section problem in Listing 5.1. Is the following computation a counterexample for the property of absence of starvation?

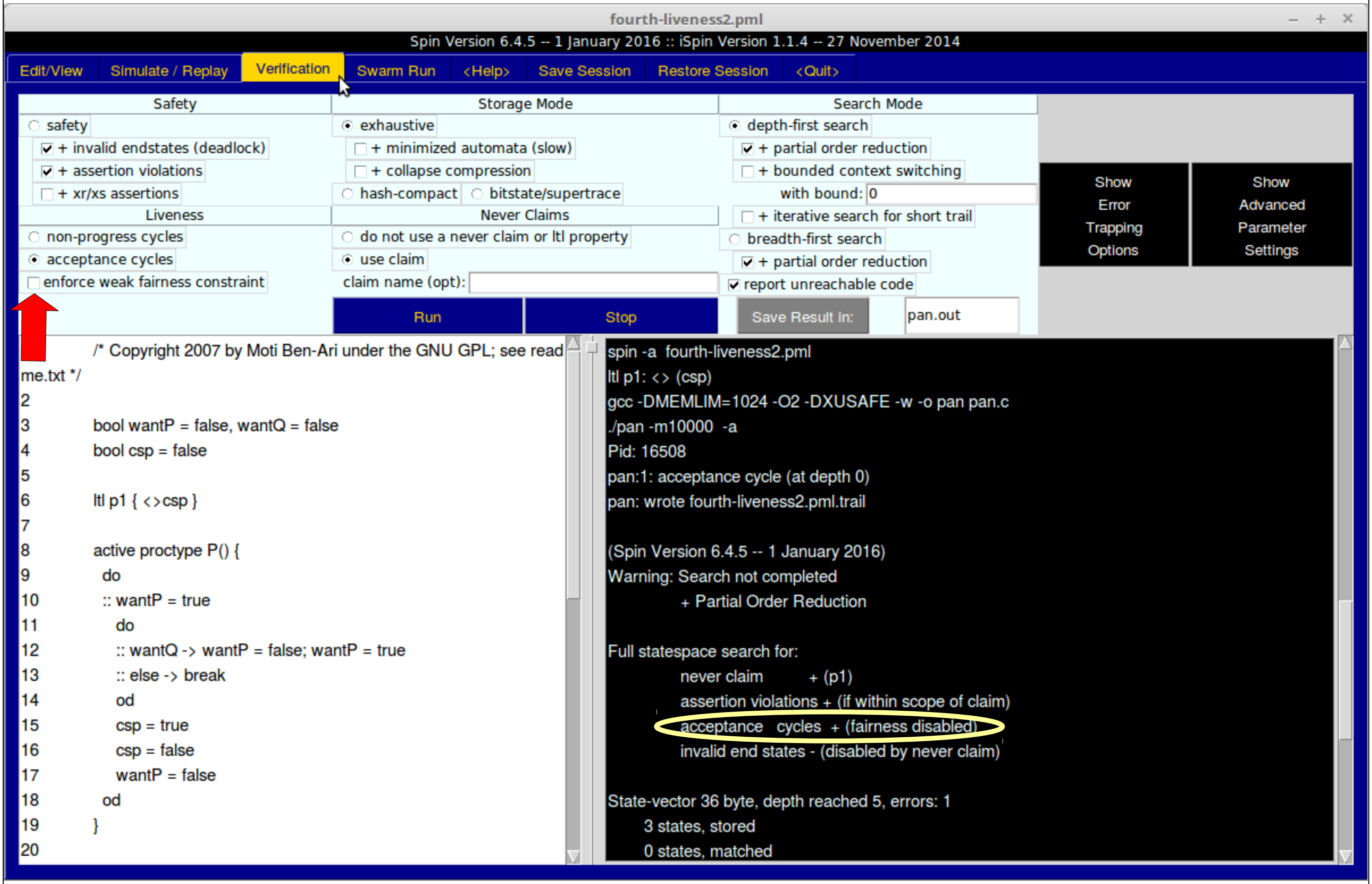
s_0 = `(5. wantP=1, 18. wantQ=1, 0, 0)` →
s_1 = `(5. wantP=1, 20. wantP,   0, 1)` →
s_2 = `(5. wantP=1, 25. wantQ=0, 0, 1)` →
s_3 = `(5. wantP=1, 18. wantQ=1, 0, 0)`

State s_3 is identical to s_0, so an infinite computation can be composed from just the three states s_0, s_1, s_2. In this computation, process Q enters its critical section repeatedly, while process P never executes any of its statements. The computation ***is*** a counterexample to a claim that `<>csp` is true, but it is unsatisfactory because it doesn't give process P a "fair" chance to try to enter its critical section.

This concept can be formalized by the following definition:

> A computation is ***weakly fair*** if and only if the following condition holds: if a statement is ***always*** executable, then it is ***eventually*** executed as part of the computation.

The computation described above is not weakly fair: Although like all assignment statements, `5. wantP=`**`true`** is always executable, it is never executed in the computation. As we have shown, absence of starvation does not, in fact, hold for the program in Listing 5.1, but it seems reasonable to requiere that only fair computations be considered as counterexamples.


fourth-liveness2.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Run
Stop
Save Result in:
Show Error Trapping Options
Show Advanced Parameter Settings

Safety
○ safety
☑ + invalid endstates (deadlock)
☑ + assertion violations
□ + xr/xs assertions

Liveness
○ non-progress cycles
● acceptance cycles
□ enforce weak fairness constraint

Storage Mode
● exhaustive
□ + minimized automata (slow)
□ + collapse compression
○ hash-compact ○ bitstate/supertrace

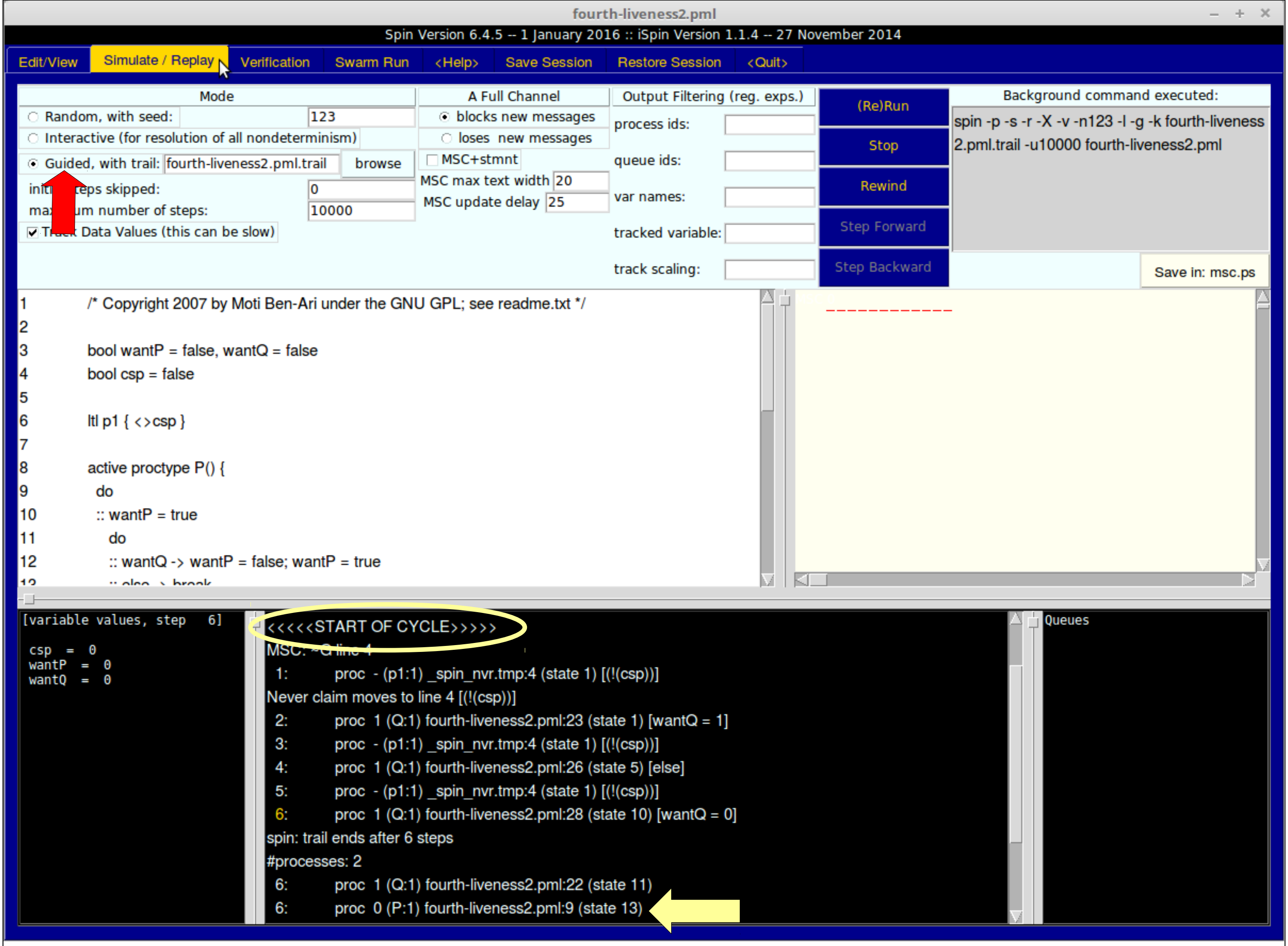
Never Claims
○ do not use a never claim or ltl property
● use claim
claim name (opt):

Search Mode
● depth-first search
☑ + partial order reduction
□ + bounded context switching
with bound: 0
□ + iterative search for short trail
○ breadth-first search
☑ + partial order reduction
☑ report unreachable code

Save Result in: pan.out

```
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

bool wantP = false, wantQ = false
bool csp = false

ltl p1 { <>csp }

active proctype P() {
  do
  :: wantP = true
     do
     :: wantQ -> wantP = false; wantP = true
     :: else -> break
     od
     csp = true
     csp = false
     wantP = false
  od
}
```

```
spin -a  fourth-liveness2.pml
ltl p1: <> (csp)
gcc -DMEMLIM=1024 -O2 -DXUSAFE -w -o pan pan.c
./pan -m10000  -a
Pid: 16508
pan:1: acceptance cycle (at depth 0)
pan: wrote fourth-liveness2.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim          + (p1)
        assertion violations + (if within scope of claim)
        acceptance   cycles  + (fairness disabled)
        invalid end states - (disabled by never claim)

State-vector 36 byte, depth reached 5, errors: 1
     3 states, stored
     0 states, matched
```



Mode
○ Random, with seed: 123
○ Interactive (for resolution of all nondeterminism)
● Guided, with trail: fourth-liveness2.pml.trail browse
initial steps skipped: 0
maximum number of steps: 10000
☑ Track Data Values (this can be slow)

A Full Channel
● blocks new messages
○ loses new messages
□ MSC+stmnt
MSC max text width 20
MSC update delay 25

Output Filtering (reg. exps.)
process ids:
queue ids:
var names:
tracked variable:
track scaling:

Background command executed:
spin -p -s -r -X -v -n123 -l -g -k fourth-liveness2.pml.trail -u10000 fourth-liveness2.pml

Save in: msc.ps

```
        /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

        bool wantP = false, wantQ = false
        bool csp = false

        ltl p1 { <>csp }

        active proctype P() {
          do
          :: wantP = true
             do
             :: wantQ -> wantP = false; wantP = true
             :: else -> break
```

```
[variable values, step   6]

 csp   =  0
 wantP  =  0
 wantQ  =  0
```

```
<<<<<START OF CYCLE>>>>>
MSC: ~G line 4
  1:    proc  - (p1:1) _spin_nvr.tmp:4 (state 1) [(!(csp))]
Never claim moves to line 4 [(!(csp))]
  2:    proc  1 (Q:1) fourth-liveness2.pml:23 (state 1) [wantQ = 1]
  3:    proc  - (p1:1) _spin_nvr.tmp:4 (state 1) [(!(csp))]
  4:    proc  1 (Q:1) fourth-liveness2.pml:26 (state 5) [else]
  5:    proc  - (p1:1) _spin_nvr.tmp:4 (state 1) [(!(csp))]
  6:    proc  1 (Q:1) fourth-liveness2.pml:28 (state 10) [wantQ = 0]
spin: trail ends after 6 steps
#processes: 2
  6:    proc  1 (Q:1) fourth-liveness2.pml:22 (state 11)
  6:    proc  0 (P:1) fourth-liveness2.pml:9 (state 13)
```

We conclude this section with an example of a program whose properties depend critically on fairness (Listing 5.2).

The assignment in process Q is always enabled, so in a weakly fair computation it will eventually be executed, causing the loop in process P to terminate.

If weak fairness is ***not*** specified, there is a nonterminating computation in which the **do**-statement is executed indefinitely.

Thus the correctness property "the program always terminates" holds if and only if computations are required to be weakly fair.

Listing 5.2. Termination under weak fairness (stopA.pml)

```
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

int n = 0
bool flag = false

active proctype p() {
  do
  ::  flag -> break
  ::  else -> n = 1 - n
  od
}

active proctype q() {
  flag = true
}
```

Listing 5.2. Termination under weak fairness (stopA.pml)

```
$ spin -run -l stopA.pml  # find non-progress cycles
pan:1: non-progress cycle (at depth 2)
pan: wrote stopA.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Partial Order Reduction

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

State-vector 36 byte, depth reached 13, errors: 1
       17 states, stored (29 visited)
       13 states, matched
       42 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
...
```

Listing 5.2. Termination under weak fairness (stopA.pml)

```
...
$ spin -t -p -g stopA.pml
starting claim 2
spin: couldn't find claim 2 (ignored)
using statement merging
  2:    proc  0 (p:1) stopA.pml:9 (state 3)     [else]
  <<<<<START OF CYCLE>>>>>
  4:    proc  0 (p:1) stopA.pml:9 (state 4)     [n = (1-n)]
                n = 1
  6:    proc  0 (p:1) stopA.pml:9 (state 3)     [else]
  8:    proc  0 (p:1) stopA.pml:9 (state 4)     [n = (1-n)]
                n = 0
 10:    proc  0 (p:1) stopA.pml:9 (state 3)     [else]
spin: trail ends after 10 steps
#processes: 2
                n = 0
                flag = 0
 10:    proc  1 (q:1) stopA.pml:14 (state 1)
 10:    proc  0 (p:1) stopA.pml:9 (state 4)
2 processes created
```

Listing 5.2. Termination under weak fairness (stopA.pml)

```
$ spin -run -l -f stopA.pml  # find non-progress cycles adding weak fairness

(Spin Version 6.4.5 -- 1 January 2016)
        + Partial Order Reduction

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

State-vector 36 byte, depth reached 13, errors: 0
       32 states, stored (40 visited)
       29 states, matched
       69 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
...
```

Section 5.6

Duality

The operators □ and ◇ are ***dual*** in a manner similar to the duality expressed by deMorgan's laws:

$$\neg(p \wedge q) \equiv (\neg p \vee \neg q), \qquad \neg(p \vee q) \equiv (\neg p \wedge \neg q).$$

Passing a negation through a unary temporal operator changes the operator to the other one:

$$\neg\Box p \equiv \Diamond\neg p, \qquad \neg\Diamond p \equiv \Box\neg p.$$

Since double negations cancel out, duality can be used to simplify formulas with temporal operators. Let ***good*** and ***bad*** be atomic propositions such that ***good*** is equivalent to **¬*bad***. Then we have the following equivalences:

$$\neg\Box\, \mathit{good} \equiv \Diamond\neg \mathit{good} \equiv \Diamond\neg\neg \mathit{bad} \equiv \Diamond \mathit{bad},$$
$$\neg\Diamond\, \mathit{good} \equiv \Box\neg \mathit{good} \equiv \Box\neg\neg \mathit{bad} \equiv \Box \mathit{bad}.$$

These make sense when read out loud: if it is false that something good is always true, then eventually something bad must happen; if it is false that something good eventually happens, then something bad always true.

It is important to get used to reasoning with the duality of the temporal operators because negations of correctness specifications are at the foundation of model checking.

Section 5.7

Verifying correctness without ghost variables

We have used ghost variables like `critical` and `csp` as proxies for control points in a Promela program. While this causes no problems in the small programs shown in the book, when modeling large systems you will want to keep the number of variables as small as possible. Ghost variables also unnecessarily complicate graphical representations of the state transition diagrams that are generated by the SpinSpider tool.

Promela supports ***remote references*** that can be used to refer to control points in correctness specifications, either directly within never claims or in LTL formulas.

For example, in a program for the critical section problem, we can replace the ghost variables by defining labels `cs` at the control points corresponding to the critical sections of the two processes and then defining a symbol that expresses mutual exclusion using remote references:

```
#define mutex !(P@cs && Q@cs)

active proctype P() {
  do
  :: wantP = true
     !wantQ
cs:  wantP = false
  od
}
/* Similarly for process Q */
```

The expresion `P@cs` returns a nonzero value if and only if the location counter of process `P` is at the control point labeled by `cs`. Mutual exclusion holds only if both `P@cs` and `Q@cs` cannot be true at the same time, expressed as `[]mutex`.

A verification run shows that this formula does indeed hold.

It is also possible to refer to the value of a local variable of a process using the syntax `process:variable`.

Section 5.8

Modeling a noncritical section

One of the correctness properties of the critical section problem is that a process be able to enter its critical section infinitely often even if another process fails in its *noncritical* section. This can be modeled in Promela by including a nondeterministic **if**-statement in a process that is allowed to fail.

The program in Listing 5.3 is a solution to the critical section problem that achieves mutual exclusion. This can be checked by verifying the safety property shown in Section 5.7: define the symbol `mutex` as `!(P@cs && Q@cs)` and verify `[]mutex`.

Listing 5.3. Modeling failure in the noncritical section (`first-ncs.pml`)

```
 /* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

 /*
    First attempt
    Simulate non-termination of non-CS
    Verify Safety - invalid end state
 */
 #define mutex !(P@cs && Q@cs)
 byte turn = 1

 active proctype P() {
   do
   ::
     if              /* NCS does nothing or halts */
     :: true
     :: true -> false
     fi
     (turn == 1)
cs:  turn = 2
   od
 }

 active proctype Q() {
   do
   ::
     (turn == 2)
cs:  turn = 1
   od
}
```

Lines 14-17 model the noncritical section: P can nondeterministically choose to do nothing (line 15) or to fail by blocking until **false** becomes true, which, of course, will never occur (line 16).

The program in Listing 5.3 is not a correct solution to the critical section problem, because if process P fails in its noncritical section (by blocking at line 16), process Q will eventually become blocked indefinitely waiting for `turn == 2` to become true (line 26).

Now add an **if**-statement like one in lines 14-17 to one of the processes of a correct solution to the critical section problem: Dekker's algorithm or Peterson's algorithm (Listing 5.4).

Define the symbol `live` as `Q@cs` and verify the absence of starvation: `[]<>live`.

Process P fails only when `wantP` is false, so process Q can continue entering its critical section infinitely often because the expression at line 23 always evaluates to true regardless of the value of the variable `last`.

Listing 5.4. Peterson's algorithm (peterson-over.pml)

```
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

#define ptry P@try
#define qcs  Q@cs
#define pcs  P@cs

bool wantP, wantQ
byte last = 1

active proctype P() {
  do
  ::  wantP = true
      last = 1
try:  (wantQ == false) || (last == 2)
cs:   wantP = false
  od
}

active proctype Q() {
  do
  ::  wantQ = true
      last = 2
try:  (wantP == false) || (last == 1)
cs:   wantQ = false
  od
}
```

Section 5.9

Advanced temporal specifications

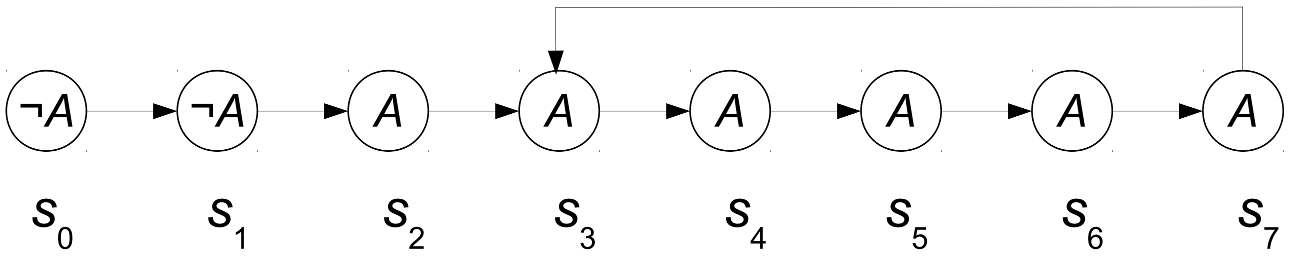
The temporal operators $\Box$ and $\Diamond$ can be applied to any formula of LTL, so that $\Box\Diamond\Diamond A$ and $\Diamond\Box\Diamond(A \wedge \Box B)$ are syntactically correct. It is beyond the scope of this book to present the deductive theory of LTL: axioms, rules of inferences, and theorems relating to properties of formulas such as associativity and commutivity (see *MLCS*, Chapter 12). We just mention two results:

- A formula with sequences of consecutive occurences of the operators $\Box$ or $\Diamond$ is equivalent to one in which the sequences are collapsed to a single occurrence of the operator. For example, $\Box\Box\Diamond\Diamond A$ is equivalent to $\Box\Diamond A$.
- A formula with any sequence of alternate occurences of the operators $\Box$ or $\Diamond$ is equivalent to one in which the sequences are collapsed into one of the two-operator sequences $\Box\Diamond$ or $\Diamond\Box$. For example, $\Diamond\Box\Diamond A$ is equivalent to $\Box\Diamond A$.

Subsection 5.9.1

Latching

The formula $\Diamond\Box A$ expresses a ***latching*** property: A may not be true initially in a computation, but eventually it becomes true and remains true:



The formula $\Diamond\Box A$ is true in s_0: Although A is not true in s_0 or s_1, it becomes true in s_2 and remains true in all subsequent states of the computation.

Latching is important because it is unusual for a property to be true initially and always; rather, some statements must be executed to make the property true, although once it becomes true, the property remains true. Latching can also express properties that relate to exceptional situations.

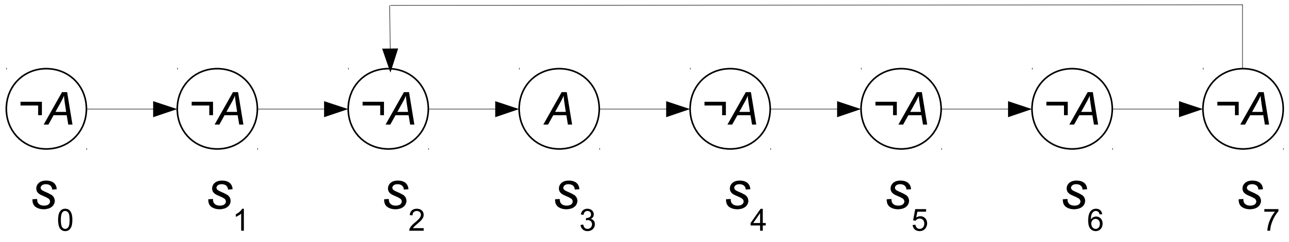
For example, suppose that a multiprocessor system is designed so that if a processor fails it automatically sets its variables to zero. Then for the program in Listing 5.1, we could claim $\Diamond \mathit{fails}_Q \rightarrow \Diamond\Box\neg \mathit{wantQ}$, that is, if ever the processor executing process `Q` fails, the value of `wantQ` is latched to false.

From this we can deduce that process `P` will not be starved even if `Q` fails because eventually the guard `wantQ` in line 7 will always be false and the `else`-alternative in line 10 can be taken.

Subsection 5.9.2

Infinitely often

The formula $\Box\Diamond A$ expresses the property that A is true *infinitely often*: A need not always be true, but at *any* state in the computation s, A will be true in s or in some state that comes after s:



It is easy to see that A is true in the states s_3, s_9, s_{15}, ..., so at any state s_i, A is true in one of the states s_i, s_{i+1}, s_{i+2}, s_{i+3}, s_{i+4}, s_{i+5}.

For solutions to the critical section problem, liveness means not just that a process can enter its critical section, but that it can enter its critical section repeteadly. This can modeled in Promela as follows. First, after setting a variable that indicates that P is in its critical section, we immediately reset it to indicate that P has left its critical section:

```
active proctype P() {
  do
  ::  /* Try to enter critical section */
      csp = true
      csp = false
      /* Leave critical section */
  od
}
```

Then – if the algorithm is free from starvation – we can verify the program for the temporal formula `[]<>csp`.

Subsection 5.9.3

Precedence

The operators □ and ◇ are unary and cannot express properties that relate two points in time, such as the *precedence* property that requires that *A* become true before *B* becomes true. This can be expressed with the binary operator $\mathcal{U}$ called *until* and written U in Spin:

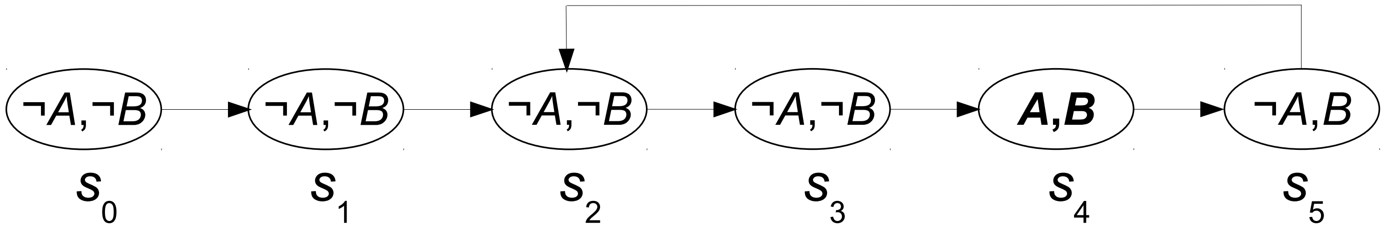
$$\neg B \mathcal{U} A.$$

Read this as: *B* remains false until *A* becomes true.
More formally:

$p\mathcal{U}q$ is true in state s_j of a computation τ if and only if there is some state s_k in τ with $k \geq i$, such that q is true in s_k, and for all s_j in τ such that $i \leq j < k$, p is true in s_j.

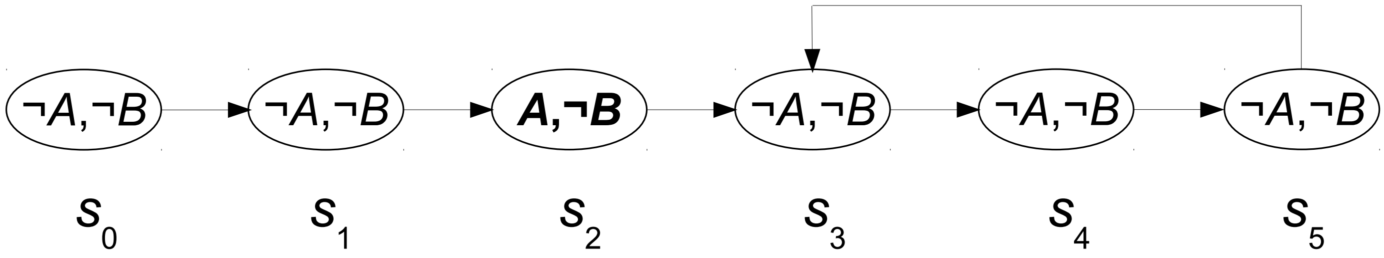
If q is already true in s_i, the second requirement is vacuous.

The formula $\neg B \mathcal{U} A$ is true in s_0 of the following computation because *B* remains false as long as *A* does; only in s_4, when *A* becomes true, does *B* also become true:



Note that *B* need not be true in s_4, because we are only interested in specifying that in remain false until *A* becomes true.

In fact, B can be false throughout the entire computation, and the truth of A beyond its first true occurrence is irrelevant; it follows that $\neg B\,\mathcal{U}\,A$ is true in s_0 of the following computation:



The operator $\mathcal{U}$ is called the *strong until* operator, because the subformula to the right of $\mathcal{U}$ is required to become true eventually. In fact $\Diamond q$ can be defined as *true* $\mathcal{U}$ q. Since *true* is trivially true, *true* $\mathcal{U}$ q is true if and only if q eventually becomes true.

There is a *weak until* operator $\mathcal{W}$ that does not require that the right subformula eventually become true. The two operators are related as follows:

$$p\,\mathcal{U}\,q \;\equiv\; p\,\mathcal{W}\,p \wedge \Diamond q, \qquad p\,\mathcal{W}\,q \;\equiv\; p\,\mathcal{U}\,q \vee \Box p.$$

Advanced: The `V` operator

Spin has an operator `V` that is defined so that `p V q` is equivalent to `!( (!p) U (!q) )`. The operator `V` is *not* the same as $\mathcal{W}$; if it were, the corresponding formula would be `!( (!q) U (!p && !q) )`.

Subsection 5.9.4

Overtaking

We will demonstrate the use of the $\mathcal{U}$ operator to specify *one-bounded overtaking* in Peterson's algorithm (Listing 5.4), a correct solution to the critical section problem. One-bounded overtaking means that if process `P` tries to enter its critical section, process `Q` can enter its critical section at most once before `P` does.

Let us define the symbols:

```
#define ptry P@try
#define qcs  Q@cs
#define pcs  P@cs
```

If process `P` is not in its critical section, it is *not* true that `csq` is false, and it is certainly not true that `csq` remains false until `P` enters its critical section. First, process `Q` may currently be in its critical section, but even if it isn't, it may *overtake* process `P` and enter its critical section first.

Listing 5.4. Peterson's algorithm (`peterson-over.pml`)

```
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

#define ptry P@try
#define qcs  Q@cs
#define pcs  P@cs

bool wantP, wantQ
byte last = 1

active proctype P() {
  do
  ::  wantP = true
      last = 1
try:  (wantQ == false) || (last == 2)
cs:   wantP = false
  od
}

active proctype Q() {
  do
  ::  wantQ = true
      last = 2
try:  (wantP == false) || (last == 1)
cs:   wantQ = false
  od
}
```

One-bounded overtaking is expressed by the LTL formula:

```
    []( ptry -> ( !qcs U ( qcs U (!qcs U pcs) ) ) )
```

A nested *until* formula of this form expresses the property that a sequence of intervals must satisfy successive subformula. The formula above expresses the property that, *always*, if process `P` is trying to enter its critical section (`ptry` is true), the computation must start with the following sequence of intervals: (a) process `Q` is not in its critical section (`!qcs`); (b) process `Q` is in its critical section (`qcs`); (c) again, process `Q` is not in its critical section (`!qcs`); and finally (d) process `P` is in its critical section (`pcs`).

According to the definition of the $\mathcal{U}$ operator, the intervals may be empty, but the correctness of this property ensures that there cannot be two *separate* intervals where `qcs` is true before the state where `pcs` becomes true.

Run a verification of the program for this formula and show that one-bounded overtaking holds.