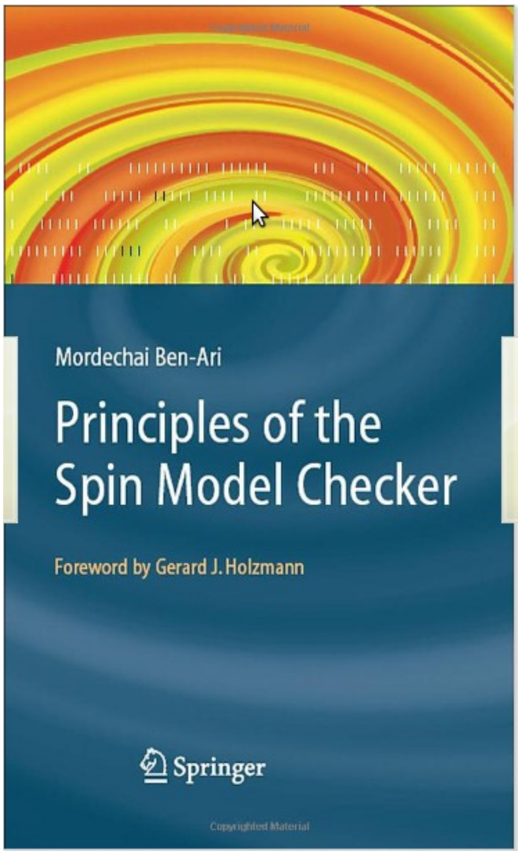


Mordechai Ben-Ari

Principles of the
Spin Model Checker

Springer, 2008

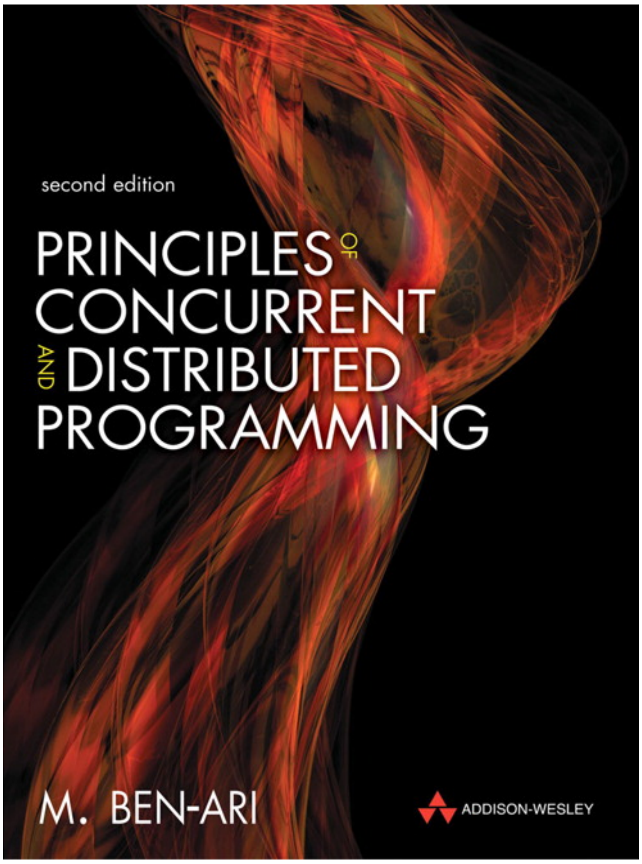
ISBN: 978-1-84628-769-5

Chapter 4

Synchronization

Section 4.1

Synchronization by blocking



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

Busy waiting with a spin lock

```
...
volatile int lock=0; { recurso libre }
...
```

Proceso 0:

```
while (lock);
lock=1;
< sección crítica >
lock=0;
< sección no crítica >
```

Proceso 1:

```
while (lock);
lock=1;
< sección crítica >
lock=0;
< sección no crítica >
```

***Note*: a lock that uses busy waiting is called a *spin lock*.**

spin_lock_v6a.pml

```
 1  /*  spin_lock_v6a.pml */
 2
 3  bit  in_use = false
 4  byte cs = 0
 5
 6  proctype P(bit i) {
 7    do
 8    ::  in_use == false ->      /* while (in_use == 1) ; i.e. busy wait */
 9          in_use = true
10          cs++
11          printf("P(%d) has entered CS\n", i)
12          assert(cs == 1)
13          cs--
14          in_use = false
15    od
16  }
17
18  init {
19    atomic { run P(0); run P(1) }
20  }
```


Simulation mode

```
$ spin spin_lock_v6a.pml
          P(0) has entered CS
              P(1) has entered CS
          P(0) has entered CS
          P(0) has entered CS
              P(1) has entered CS
          P(0) has entered CS
          P(0) has entered CS
              P(1) has entered CS
          P(0) has entered CS
              P(1) has entered CS
              P(1) has entered CS
          P(0) has entered CS
spin: spin_lock_v6a.pml:12, Error: assertion violated
spin: text of failed assertion: assert((cs==1))
#processes: 3
                in_use = 1
                cs = 2
 91:    proc  2 (P:1) spin_lock_v6a.pml:12 (state 5)
 91:    proc  1 (P:1) spin_lock_v6a.pml:12 (state 5)
 91:    proc  0 (:init::1) spin_lock_v6a.pml:20 (state 4) <valid end state>
3 processes created
```

spin_lock_v6b.pml

```
/*  spin_lock_v6b.pml */

bit  in_use = false
byte cs = 0

proctype P(bit i) {
  do
  :: atomic {
       in_use == false ->      /* while (in_use == 1) ; i.e. busy wait */
         in_use = true
     }
     cs++
     printf("P(%d) has entered CS\n", i)
     assert(cs == 1)
     cs--
     in_use = false
  od
}

init {
  atomic { run P(0); run P(1) }
}
```


Verification mode
(long version)

```
$ ls -l spin_lock_v6*
-rw-r--r-- 1 vk vk 341 mar 27 22:48 spin_lock_v6a.pml
-rw-r--r-- 1 vk vk 356 mar 27 22:48 spin_lock_v6b.pml

$ spin -a spin_lock_v6b.pml

$ ls -l {spin_lock_v6*,pan*}
-rw-r--r-- 1 vk vk    719 sep 15 10:49 pan.b
-rw-r--r-- 1 vk vk 329266 sep 15 10:49 pan.c
-rw-r--r-- 1 vk vk  16358 sep 15 10:49 pan.h
-rw-r--r-- 1 vk vk   2991 sep 15 10:49 pan.m
-rw-r--r-- 1 vk vk  56161 sep 15 10:49 pan.p
-rw-r--r-- 1 vk vk  18956 sep 15 10:49 pan.t
-rw-r--r-- 1 vk vk    341 mar 27 22:48 spin_lock_v6a.pml
-rw-r--r-- 1 vk vk    356 mar 27 22:48 spin_lock_v6b.pml

$ gcc pan.c -o pan

$ ls -l pan*
-rwxr-xr-x 1 vk vk  95376 sep 15 10:55 pan
-rw-r--r-- 1 vk vk    719 sep 15 10:49 pan.b
...
```

Verification mode
(short version)

```
$ rm pan*                    # sin espacio antes de *!

$ ls -l spin_lock_v6*
-rw-r--r-- 1 vk vk 341 mar 27 22:48 spin_lock_v6a.pml
-rw-r--r-- 1 vk vk 356 mar 27 22:48 spin_lock_v6b.pml

$ spin -run spin_lock_v6b.pml

(Spin Version 6.4.8 -- 2 March 2018)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 28 byte, depth reached 7, errors: 0
       12 states, stored
        2 states, matched
       14 transitions (= stored+matched)
        1 atomic steps
hash conflicts:         0 (resolved)

...
```


Verification mode
(short version)

```
...
Stats on memory usage (in Megabytes):
    0.001       equivalent memory usage for states (stored*(State-vector + overhead))
    0.291       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


unreached in proctype P
        spin_lock_v6b.pml:18, state 12, "-end-"
        (1 of 12 states)
unreached in init
        (0 of 4 states)

pan: elapsed time 0 seconds

$ ls -l {spin_lock_v6*,pan}
-rwxr-xr-x 1 vk vk  78968 sep 15 11:04 pan
-rw-r--r-- 1 vk vk    341 mar 27 22:48 spin_lock_v6a.pml
-rw-r--r-- 1 vk vk    356 mar 27 22:48 spin_lock_v6b.pml
```

M. Ben-Ari: First attempt (Strict alternation)

Cada proceso tiene su turno.

Fig. 3.1.

M. Ben-Ari: First attempt (Strict alternation)

```
volatile int turn=1; { turno del proceso P1 }
...
```

***Proceso 1*:**

```
while (TRUE) {
  while (turn != 1);
  critical_section();
  turn=2;
  noncritical();
}
```

***Proceso 2*:**

```
while (TRUE) {
  while (turn != 2);
  critical_section();
  turn=1;
  noncritical();
}
```

Exclusión mutua se cumple. *Deadlock* es imposible.
Espera limitada se cumple. Progreso no se cumple.

Si el P1 necesita entrar en su CS 100 veces al día, y el P2 necesita entrar en su CS una vez al día, entonces el P1 no podrá hacerlo.

Otro caso: el P2 se encontró con un oso polar (en su sección no crítica) y terminó. El P1 se queda desesperado en el estado *deadlocked*.

first_v6a.pml

```
/*  first_v6a.pml */

bit  turn = 0
byte cs = 0

proctype P(bit i) {
  do
  :: turn == i ->  /* while (turn != i) ; i.e. busy wait */
       cs++
       printf("P(%d) has entered CS\n", i)
       assert(cs == 1)
       cs--
       turn = 1 - i
  od
}

init {
  atomic { run P(0); run P(1) }
}
```


Verification mode

```
$ spin -run first_v6a.pml
...
State-vector 28 byte, depth reached 13, errors: 0
...

$ spin -u50 first_v6a.pml
          P(0) has entered CS
              P(1) has entered CS
          P(0) has entered CS
              P(1) has entered CS
          P(0) has entered CS
              P(1) has entered CS
          P(0) has entered CS
-----------
depth-limit (-u50 steps) reached
#processes: 3
                turn = 1
                cs = 0
 50:    proc  2 (P:1) first_v6a.pml:7 (state 7)
 50:    proc  1 (P:1) first_v6a.pml:15 (state 8)
 50:    proc  0 (:init::1) first_v6a.pml:19 (state 4) <valid end state>
3 processes created
```

first-fatal_v6a.pml

```
1  /*  first-fatal_v6a.pml */
2
3  bit  turn = 0
4  byte cs = 0
5
6  proctype P(bit i) {
7    do
8    ::  turn == i ->    /* while (turn != i) ; i.e. busy wait */
9          cs++
10         printf("P(%d) has entered CS\n", i)
11         assert(cs == 1)
12         cs--
13         turn = 1 - i
14   ::  i == 1 ->
15         skip
16   ::  i == 1 ->
17         break
18   od
19 }
20
21 init {
22   atomic { run P(0); run P(1) }
23 }
```


Simulation mode

```
$ spin -u100 first-fatal_v6a.pml
          P(0) has entered CS
              P(1) has entered CS
          P(0) has entered CS
              P(1) has entered CS
          P(0) has entered CS
              P(1) has entered CS
          P(0) has entered CS
              P(1) has entered CS
          P(0) has entered CS
      timeout
#processes: 2
                turn = 1
                cs = 0
 79:    proc  1 (P:1) first-fatal_v6a.pml:7 (state 11)
 65:    proc  0 (:init::1) first-fatal_v6.pml:23 (state 4) <valid end state>
3 processes created
```

Verification mode

```
$ spin -run first-fatal_v6a.pml
pan:1: invalid end state (at depth 11)
pan: wrote first-fatal_v6a.pml.trail

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 28 byte, depth reached 12, errors: 1
...
```


Verification mode

```
$ spin -run -r first-fatal_v6a.pml
  1:    proc  0 (:init:) first-fatal_v6a.pml:22 (state 3)      [(run P(0))]
  2:    proc  0 (:init:) first-fatal_v6a.pml:22 (state 2)      [(run P(1))]
  3:    proc  2 (P) first-fatal_v6a.pml:8 (state 11)    [((i==1))]
  4:    proc  1 (P) first-fatal_v6a.pml:8 (state 11)    [((turn==i))]
  5:    proc  2 (P) first-fatal_v6a.pml:15 (state 8)    [(1)]
  6:    proc  2 (P) first-fatal_v6a.pml:8 (state 11)    [((i==1))]
  7:    proc  2 (P) -:0 (state 0)       [-end-]
  8:    proc  1 (P) first-fatal_v6a.pml:9 (state 2)     [cs = (cs+1)]
P(0) has entered CS
  9:    proc  1 (P) first-fatal_v6a.pml:10 (state 3)    [printf('P(%d) has entered
CS\n',i)]
 10:    proc  1 (P) first-fatal_v6a.pml:11 (state 4)    [assert((cs==1))]
 11:    proc  1 (P) first-fatal_v6a.pml:12 (state 5)    [cs = (cs-1)]
 12:    proc  1 (P) first-fatal_v6a.pml:13 (state 6)    [turn = (1-i)]
spin: trail ends after 12 steps
#processes 2:
 12:    proc 0 (:init:)  first-fatal_v6a.pml:23 (state  4)
                -end-
 12:    proc 1 (P)  first-fatal_v6a.pml:8 (state 11) (invalid end state)
                ((turn==i))
                ((i==1))
                ((i==1))
global vars:
        bit     turn:    1
        byte    cs:      0
local vars proc 1 (P):
        bit     i:       0
```

pan option **-r**: read and execute trail

M. Ben-Ari: Second attempt

Para remediar el problema de la "solución" anterior daremos a cada proceso su propia llave de acceso a su sección crítica (y ya no importa el problema con el oso polar).

***c*[2] = 1 significa que el proceso 2 no está en su sección crítica;**
el proceso 1 verifica *c*[2]; marca *c*[1] = 0 y entra en su sección crítica;
al salir, establece *c*[1] = 1.

Fig. 3.3.

M. Ben-Ari: Second attempt

```
volatile int c1=1, c2=1; { ninguno en su SC }

...
```

Proceso 1:

```
while (TRUE) {
  while (c2 == 0);
  c1 = 0;
  cs1();
  c1 = 1;
  ncs1();
}
```

Proceso 2:

```
while (TRUE) {
  while (c1 == 0);
  c2 = 0;
  cs2();
  c2 = 1;
  ncs2();
}
```

second_v6a.pml

```
/*  second_v6a.pml */

bit  c[2] = 1
byte cs = 0

proctype P(bit i) {
  do
  ::  c[1-i] == 1 ->
        c[i]=0
        cs++
        printf("P(%d) has entered CS\n", i)
        assert(cs == 1)
        cs--
        c[i]=1
  od
}

init {
  atomic { run P(0); run P(1) }
}
```


Verification mode

```
$ spin -run second_v6a.pml
pan:1: assertion violated (cs==1) (at depth 11)
pan: wrote second_v6a.pml.trail

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 36 byte, depth reached 15, errors: 1
...
```

Verification mode

```
$ spin -run -r second_v6a.pml
  1:    proc  0 (:init:) second_v6a.pml:19 (state 3)    [(run P(0))]
  2:    proc  0 (:init:) second_v6a.pml:19 (state 2)    [(run P(1))]
  3:    proc  2 (P) second_v6a.pml:8 (state 8)  [((c[(1-i)]==1))]
  4:    proc  1 (P) second_v6a.pml:8 (state 8)  [((c[(1-i)]==1))]
  5:    proc  2 (P) second_v6a.pml:9 (state 2)  [c[i] = 0]
  6:    proc  2 (P) second_v6a.pml:10 (state 3) [cs = (cs+1)]
P(1) has entered CS
  7:    proc  2 (P) second_v6a.pml:11 (state 4) [printf('P(%d) has entered CS\n',i)]
  8:    proc  2 (P) second_v6a.pml:12 (state 5) [assert((cs==1))]
  9:    proc  1 (P) second_v6a.pml:9 (state 2)  [c[i] = 0]
 10:    proc  1 (P) second_v6a.pml:10 (state 3) [cs = (cs+1)]
P(0) has entered CS
 11:    proc  1 (P) second_v6a.pml:11 (state 4) [printf('P(%d) has entered CS\n',i)]
pan:1: assertion violated (cs==1) (at depth 12)
spin: trail ends after 12 steps
...
```

pan option **-r**: read and execute trail

Verification mode

```
...
#processes 3:
 12:    proc 0 (:init:)  second_v6a.pml:20 (state  4)
                -end-
 12:    proc 1 (P)  second_v6a.pml:12 (state  5) (invalid end state)
                assert((cs==1))
 12:    proc 2 (P)  second_v6a.pml:13 (state  6) (invalid end state)
                cs = (cs-1)
global vars:
        bit     c[0]:     0
        bit     c[1]:     0
        byte    cs:       2
local vars proc 1 (P):
        bit     i:        0
local vars proc 2 (P):
        bit     i:        1
```

M. Ben-Ari: Third attempt

Analizando el fracaso del segundo intento, se puede anotar que el Proceso 1, al averiguar que el Proceso 2 no está en su sección crítica, de inmediato toma la decisión de entrar en su sección crítica.

De esa manera, en el instante cuando el Proceso 1 (*P1*) sale de *while*, él realmente ya está en su sección crítica. Esto contradice con la intención inicial de indicar con *c1* = 0 que el *P1* está en su sección crítica, porque entre las sentencias *while* y de asignación puede suceder una espera arbitrariamente larga.

En esta versión, la asignación *c1* = 0 se hace antes de la verificación de *c2*:

M. Ben-Ari: Third attempt

```
volatile int c1=1, c2=1; { ninguno en su SC }
...
```

Proceso 1:

```
while (TRUE){
  c1 = 0;
  while (c2 == 0);
  cs1();
  c1 = 1;
  ncs1();
}
```

Proceso 2:

```
while (TRUE){
  c2 = 0;
  while (c1 == 0);
  cs2();
  c2 = 1;
  ncs2();
}
```

Desgraciadamente, este programa lleva fácilmente al *deadlock* del sistema como se ve en este guión:

	c1	c2
Inicialmente	1	1
P1 establece c1	0	1
P2 establece c2	0	0
P1 verifica c2	0	0
P2 verifica c1	0	0

Listing 4.1. Synchronization by busy-waiting (`third-do_v6.pml`)

```
bool wantP = false, wantQ = false

active proctype P() {
  do
  ::
    printf("Noncritical section P\n")
    wantP = true
    do
    :: !wantQ -> break
    :: else -> skip
    od
    printf("Critical section P\n")
    wantP = false
  od
}

active proctype Q() {
  do
  ::
    printf("Noncritical section Q\n")
    wantQ = true
    do
    :: !wantP -> break
    :: else -> skip
    od
    printf("Critical section Q\n")
    wantQ = false
  od
}
```


Recall (Section 1.6) that an **if**-statement contains a set of alternatives that start with expressions called guards.

An alternative is ***executable*** if its guard evaluates to **true** (or 1, which is the same). The choice of the alternative to execute is made nondeterministically among the executable alternatives.

If no guards evaluate to true, the **if**-statement itself is not executable. Similarly, in a **do**-statement, if the guards of all alternatives evaluate to false, the statement is not executable and the process is blocked.

```
do
:: !wantQ -> break
:: else -> skip
od
```

```
do
:: !wantQ -> break
od
```

Listing 4.1a. Synchronization by blocking (`third-do-blocking_v6.pml`)

```
bool wantP = false, wantQ = false

active proctype P() {
  do
  ::
    printf("Noncritical section P\n")
    wantP = true
    do
    :: !wantQ -> break
    od
    printf("Critical section P\n")
    wantP = false
  od
}

active proctype Q() {
  do
  ::
    printf("Noncritical section Q\n")
    wantQ = true
    do
    :: !wantP -> break
    od
    printf("Critical section Q\n")
    wantQ = false
  od
}
```


To say that a process is blocked means that in simulation mode Spin will not choose the next statement to execute from that process.

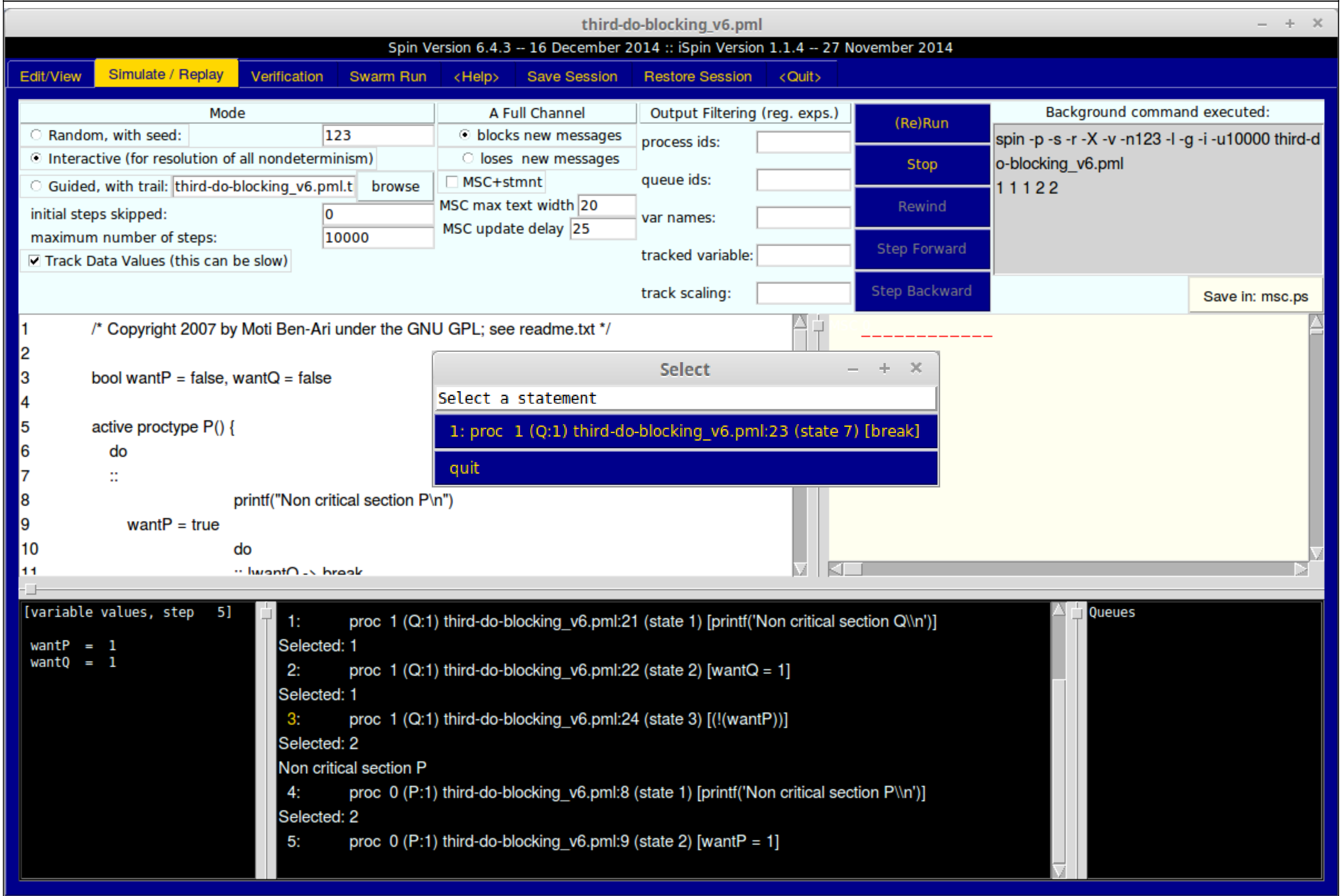
In verification mode it means that Spin will not continue the search for a counterexample from this state by looking for states that can be reached by executing a statement from the process.

Hopefully, a subsequent execution of statements from other processes will *unblock* the blocked process, enabling it to continue executing in simulation mode, and in verification mode, enabling the verifier to search for states reachable by executing a statement from the process.

Check this behavior by running an interactive simulation of the program.

Execute statements of the program until a state is reached in which process **P** is blocked because `wantQ` is true; in this state you will not be allowed to choose to execute a statement from process **P**.

Now choose to execute statements from **Q** until the statement `wantQ = false` is executed, enabling the execution of the a statement from **P**.



Section 4.2

Executability of statements

There is something rather strange about the construct:

```
do
:: !wantQ -> break
od
```

Either `wantQ` is false and the **`break`** causes the loop to be left, or it is true and the process blocks; when it is unblocked the process can leave the loop. In no case is there any "looping", so the **`do`**-statement is superfluous.
In Promela it is possible to block on a simple statement, not just on a compound statement.

```
!wantQ
```

An expression statement is *executable* if and only if it evaluates to true, in this case if the value of `wantQ` is false.

Listing 4.2. Synchronization with deadlock (`third-deadlock_v6.pml`)

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  ::
    printf("Noncritical section P\n")
    wantP = true
    !wantQ
    printf("Critical section P\n")
    wantP = false
  od
}

active proctype Q() {
  do
  ::
    printf("Noncritical section Q\n")
    wantQ = true
    !wantP
    printf("Critical section Q\n")
    wantQ = false
  od
}
```


Section 4.3

State transition diagrams

Listing 4.3. Abbreviated solution for the critical section problem

```
bool wantP = false, wantQ = false

active proctype P() {
  do :: wantP = true
        !wantQ
        wantP = false
  od
}

active proctype Q() {
  do :: wantQ = true
        !wantP
        wantQ = false
  od
}
```


Recall (Section 2.1) that a *state* of a program is a set of values of the variables and the location counters, and consider a program with two processes *p* and *q* that have s_p and s_q statements, respectively, and two variables *x* and *y* that range over v_x and v_y values, respectively.

The number of possible states that can appear in computations of the program is

$$s_p \cdot s_q \cdot v_x \cdot v_y \, .$$

For example, the program in Listing 4.3 has $3 \cdot 3 \cdot 2 \cdot 2 = 36$ possible states.

However, not every possible state is *reachable* from the initial state during a computation of the program.

In particular, a solution to the critical section problem is correct only if there are possible states that are *not* reachable, namely, states where the location counters of both processes are in their critical sections, thus falsifying the requirement of mutual exclusion.

In principle, the set S of reachable states of a program is easily constructed:

1. Let $S = \{s_0\}$, where s_0 is the initial state; mark s_0 as *unexplored*.

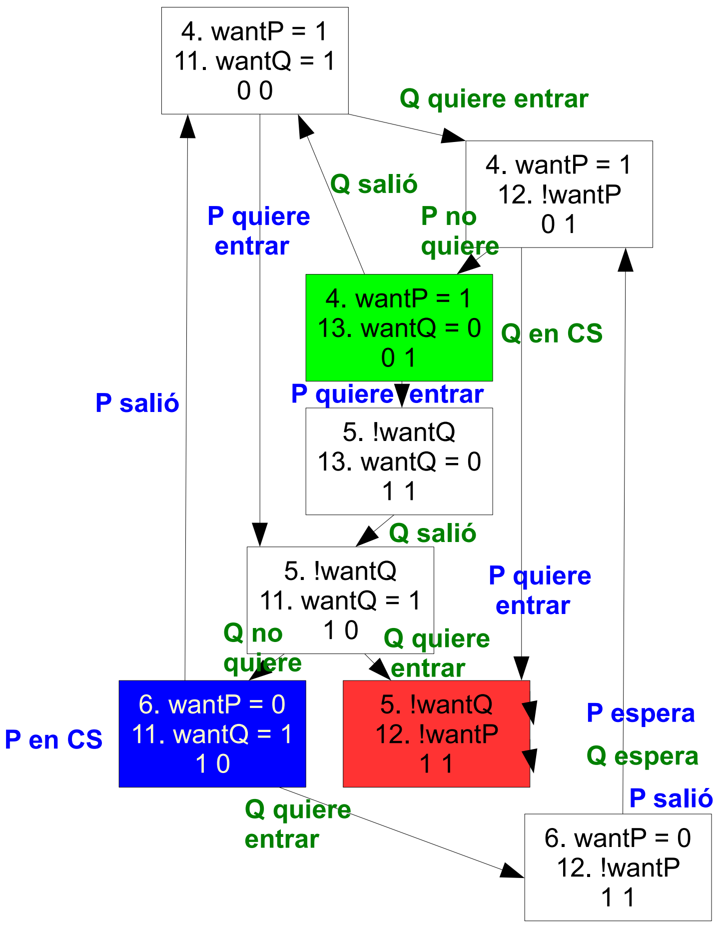
2. For each unexplored state $s \in S$, let t be a state that results from executing an executable statement in state s; if $t \notin S$, add t to S and mark it unexplored. If no such states exist, mark s as *explored*.

3. Terminate when all states in S are marked explored.

The reachable states of a concurrent program can be visualized as a connected directed graph called a *state transition diagram*.

The nodes of the diagram are the reachable states and an edge exists from state *s* to state *t* if and only if there is a statement whose execution in *s* leads to *t*.

Fig. 4.1. State diagram for the program in Listing 4.3



The number of reachable states (8) is *much* less than the number of possible states (36).

The program in Listing 4.3 is an abbreviated version of the program in Listing 4.2. The **`printf`** statements representing the critical and noncritical sections have been removed to obtain a more concise diagram. A **`printf`** statement is always executable and does not change the variables of the program, so if a state exists with a location counter before a print statement, there also exists a state with the location counter after the statement and with the same values for the variables. The same correctness specifications will thus be provable whether the print statements appear or not.

Consider the mutual exclusion property for the program in Listing 4.2. It holds if and only no state `(11. printf(P), 22. printf(Q), x, y)` is reachable for arbitrary `x` and `y`.

Therefore, mutual exclusion holds if and only there is no state `(12. wantP=0, 23. wantQ=0, x, y)`.

Clearly, then, mutual exclusion holds if and only if, in the abbreviated program, a state of the form `(6. wantP=0, 13. wantQ=0, x, y)` is not reachable.

A quick glance at the diagram in Figure 4.1 shows that no such state exists, so mutual exclusion must hold.

The program is not free from deadlock. The state `(5. !wantQ, 12. !wantP, 1, 1)` is reachable and in that state both processes are trying to enter their critical sections, but neither can succeed.

Simulation mode

```
$ spin third-abbrev_v6.pml
      timeout
#processes: 2
                wantP = 1
                wantQ = 1
  2:    proc  1 (Q:1) third-abbrev_v6.pml:14 (state 2)
  2:    proc  0 (P:1) third-abbrev_v6.pml:7 (state 2)
2 processes created


$ spin third-abbrev.pml
      timeout
#processes: 2
                wantP = 1
                wantQ = 1
 54:    proc  1 (Q:1) third-abbrev.pml:14 (state 2)
 54:    proc  0 (P:1) third-abbrev.pml:7 (state 2)
2 processes created
```


Verification mode

```
$ spin -run third-abbrev_v6.pml
pan:1: invalid end state (at depth 4)
pan: wrote third-abbrev_v6.pml.trail

(Spin Version 6.4.6 -- 2 December 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 20 byte, depth reached 5, errors: 1
...
```

Section 4.4

Atomic sequences of statements

Listing 4.4. Atomic sequences of statements (`third-atomic_v6.pml`)

```
bool wantP = false, wantQ = false          /* starvation is possible */

active proctype P() {
  do
  :: printf("Noncritical section P\n")
     atomic {
       !wantQ
       wantP = true
     }
     printf("Critical section P\n")
     wantP = false
  od
}

active proctype Q() {
  do
  :: printf("Noncritical section Q\n")
     atomic {
       !wantP
       wantQ = true
     }
    printf("Critical section Q\n")
    wantQ = false
  od
}
```

Simulation mode

```
$ spin -u30 third-atomic_v6.pml
          Noncritical section Q
          Critical section Q
          Noncritical section Q
          Critical section Q
      Noncritical section P
      Critical section P
      Noncritical section P
          Noncritical section Q
          Critical section Q
          Noncritical section Q
      Critical section P
-------------
depth-limit (-u30 steps) reached
#processes: 2
                wantP = 0
                wantQ = 0
 30:   proc  1 (Q:1) third-atomic_v6.pml:20 (state 4)
 30:   proc  0 (P:1) third-atomic_v6.pml:15 (state 8)
2 processes created
```


Verification mode

```
$ spin -run third-atomic_v6.pml

...
(Spin Version 6.4.8 -- 2 March 2018)
        + Partial Order Reduction
...
State-vector 20 byte, depth reached 6, errors: 0
...
```

Subsection 4.4.1

`d_step` and `atomic`

In Section 3.7 we mentioned that there are two constructs in Promela for specifying that a sequence of statements must be executed atomically: **`d_step`** and **`atomic`**.

The advantage of **`d_step`** is that it is extremely efficient because the statements of the sequence are executed or verified as a single step in a fully deterministic manner. However, there are three limitations on **`d_step`**:

- Except for the first statement in the sequence (the guard), statements cannot block.
- It is illegal to jump into the sequence or out of it using **`goto`** or **`break`**.

- Nondeterminism is always resolves by choosing the first true alternative in a guarded command. For example, if `a` equals `b` in the following code, the value of branch will always equal 1:

```
d_step {
  if
  :: a >= b -> max = a; branch = 1
  :: b >= a -> max = b; branch = 2
  fi
}
```

`d_step` is usually reserved for fragments of sequential code, while **`atomic`** is preferred for implementing synchronization primitives.

Listing 4.5. Unreliable relay (`relay_v6_atomic.pml`)

```
byte input, output, i

active proctype Source() {
  for (i : 1 .. 10) {
    input == 0 /* Wait until empty */
    input = i
  }
}

active proctype Relay() {
  do
  :: atomic {      /* replace atomic by d_step */
       input != 0
       output == 0
       if
       :: output = input
       :: skip    /* Drop input data */
       fi
     }
     input = 0
  od
}

...
```

Listing 4.5. Unreliable relay (relay_v6_atomic.pml)

```
...
active proctype Destination() {
  do
  ::   output != 0 /* Wait until full */
       printf("Output = %d\n", output)
       output = 0
  od
}
```


Simulation mode

```
$ spin relay_v6_atomic.pml
            Output = 2
            Output = 3
            Output = 4
            Output = 5
            Output = 6
            Output = 10
      timeout
#processes: 3
              input = 0
              output = 0
              i = 11
128:   proc  2 (Destination:1) relay_v6_atomic.pml:27 (state 4)
128:   proc  1 (Relay:1) relay_v6_atomic.pml:13 (state 9)
128:   proc  0 (Source:1) relay_v6_atomic.pml:10 (state 11) <valid end
state>
3 processes created
```

Listing 4.5. Unreliable relay (`relay_v6_atomic.pml`)

Process `Relay` transfers data from the `Source` to the `Destination`. It nondeterministically either transfers the value from `input` to `output` or it ignores the data.

If **`atomic`** is replaced by **`d_step`**, two problems occur.

First, since nondeterminism is resolved deterministically in favor of the first alternative, no data are ever dropped at line 18, and the output sequence is always the same as the input sequence.

Second, it is not legal to block at line 15 which is within the **`d_step`** sequence; this can be modeled, for example, by using a `for`-loop with an upper bound less than ten instead of the nonterminating **`do`**-statement in `Destination`.

Listing 4.5. Unreliable relay (`relay_v6_d_step.pml`)

```
 3  byte input, output, i
 4
 5  active proctype Source() {
 6    for (i : 1 .. 10) {
 7      input == 0 /* Wait until empty */
 8      input = i
 9    }
10  }
11
12  active proctype Relay() {
13    do
14    :: d_step {
15         input != 0
16         output == 0
17         if
18         :: output = input
19         :: skip    /* Drop input data */
20         fi
21       }
22       input = 0
23    od
24  }
25
...
```

Listing 4.5. Unreliable relay (`relay_v6_d_step.pml`)

```
...
active proctype Destination() {
  do
  ::   output != 0 /* Wait until full */
       printf("Output = %d\n", output)
       output = 0
  od
}
```


Simulation mode

```
$ spin relay_v6_d_step.pml
              Output = 1
              Output = 2
              Output = 3
              Output = 4
              Output = 5
              Output = 6
              Output = 7
              Output = 8
spin: relay_v6_d_step.pml:16, Error: stmnt in d_step blocks
spin: relay_v6_d_step.pml:16, Error: stmnt in d_step blocks
              Output = 9
spin: relay_v6_d_step.pml:16, Error: stmnt in d_step blocks
              Output = 10
      timeout
#processes: 3
               input = 0
               output = 0
               i = 11
144:   proc  2 (Destination:1) relay_v6_d_step.pml:27 (state 4)
144:   proc  1 (Relay:1) relay_v6_d_step.pml:13 (state 9)
144:   proc  0 (Source:1) relay_v6_d_step.pml:10 (state 11) <valid end
state>
3 processes created
```

An unreliable relay can also be modeled using channels:

```
active proctype Relay() {
  byte i
  do
  :: atomic {
       input ? i
       if
       :: output ! i
       :: skip
       fi
     }
  od
}
```

Again, changing **atomic** to **d_step** cancels the nondeterministic selection of an alternative and can cause an error at the output statement `output ! i` if the channel is full or if a rendezvous channel is used and the process `Destination` is not ready.

M. Ben-Ari: Fourth attempt

```
volatile int c1=1, c2=1; { ninguno en su SC }
...
```

Proceso 1:

```
while (TRUE){
  c1 = 0;
  while (c2 == 0) {
    c1 = 1;
    /* espera */
    c1 = 0;
  }
  cs1();
  c1 = 1;
  ncs1();
}
```

Proceso 2:

```
while (TRUE){
  c2 = 0;
  while (c1 == 0) {
    c2 = 1;
    /* espera */
    c2 = 0;
  }
  cs2();
  c2 = 1;
  ncs2();
}
```

M. Ben-Ari: Fourth attempt

Es posible el siguiente guión:

	c1	c2
Inicialmente	1	1
P1 establece c1	0	1
P2 establece c2	0	0
P1 verifica c2	0	0
P2 verifica c1	0	0
P1 establece c1	1	0
P2 establece c2	1	1
P1 establece c1	0	1
P2 establece c2	0	0

. . .

No se puede garantizar la espera limitada: *lockout*, *starvation*.

The first solution: Algorithm of T. Dekker (1965)

En 1965, E. Dijkstra presentó el algoritmo del matemático holandés Th. J. Dekker que es una ingeniosa combinación del primer y del cuarto intentos.

Dijkstra, E. *Cooperating Secuencial Processes*. Technological University, Eindhoven, The Netherlands, 1965. (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press. New York, NY 1996.)

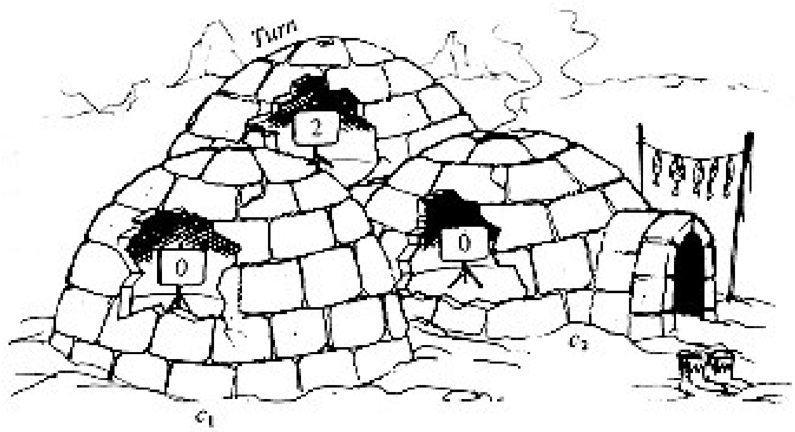


Fig. 3.8.

The first solution: Algorithm of T. Dekker (1965)

```
volatile int c1=1, c2=1; { ninguno en su SC }
volatile int turn=1;
...
```

Proceso 1:

```
while (TRUE){
  c1 = 0;
  while (c2 == 0)
    if (turn == 2) {
      c1 = 1;
      while (turn == 2);
      c1 = 0;
    };
  cs1();
  turn = 2;
  c1 = 1;
  ncs1();
}
```

Proceso 2:

```
while (TRUE){
  c2 = 0;
  while (c1 == 0)
    if (turn == 1) {
      c2 = 1;
      while (turn == 1);
      c2 = 0;
    };
  cs2();
  turn = 1;
  c2 = 1;
  ncs2();
}
```


dekker.pml

```
/* Dekker's algorithm */
bool wantp = false, wantq = false
byte turn = 1
bool csp = false, csq = false

ltl { []<>csp && []<>csq }

active proctype p() {
    do
    ::  wantp = true
        do
        :: !wantq -> break
        :: else ->
            if
            :: (turn == 1)
            :: (turn == 2) ->
                wantp = false
                (turn == 1)
                wantp = true
            fi
        od
        csp = true
        assert (!(csp && csq))
        csp = false
        wantp = false
        turn = 2
    od
}

```

dekker.pml

```
30  active proctype q() {
31      do
32      ::  wantq = true
33          do
34          :: !wantp -> break
35          :: else ->
36              if
37              :: (turn == 2)
38              :: (turn == 1) ->
39                  wantq = false
40                  (turn == 2)
41                  wantq = true
42              fi
43          od
44          csq = true
45          assert (!(csp && csq))
46          csq = false
47          wantq = false
48          turn = 1
49      od
50  }
```


Simulation mode

```
$ spin -u1000 dekker.pml
ltl ltl_0: ([] (<> (csp))) && ([] (<> (csq)))
-------------
depth-limit (-u1000 steps) reached
#processes: 2
               wantp = 0
               wantq = 1
               turn = 1
               csp = 0
               csq = 0
1000:  proc  1 (q:1) dekker.pml:44 (state 13)
1000:  proc  0 (p:1) dekker.pml:18 (state 8)
2 processes created
```

Verification mode

```
$ spin -run -noclaim dekker.pml
ltl ltl_0: ([] (<> (csp))) && ([] (<> (csq)))

(Spin Version 6.4.8 -- 2 March 2018)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 20 byte, depth reached 74, errors: 0
...
```


Verification mode

```
$ spin -run dekker.pml
ltl ltl_0: ([] (<> (csp))) && ([] (<> (csq)))
pan:1: acceptance cycle (at depth 6)
pan: wrote dekker.pml.trail

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             + (ltl_0)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 28 byte, depth reached 10, errors: 1
...
```

The simple solution: Peterson's algorithm (1981)

En 1981, G. L. Peterson presentó una elegante y muy sencilla solución para lograr la exclusión mutua.

Peterson, G. *Myths About the Mutual Exclusion Problem*. Information Processing Letters, June 1981.

```
#define FALSE 0
#define TRUE  1
#define N     2            /* número de procesos */

int turn;                  /* ¿a quién le toca?  */
int interested[N];        /* inicialmente son 0 (FALSE)*/
```


The simple solution: Peterson's algorithm (1981)

```
void enter_region(int process);     /* proceso 0 ó 1 */
{
  int other;                    /* número del otro proceso */

  other = 1 - process;
  interested[process] = TRUE;      /* mostrar interés */
  turn = process;                        /* tomar turno */
  while(turn == process &&
        interested[other] == TRUE);
}


void leave_region(int process)
{
  interested[process] = FALSE;
}
```

peterson.pml

```
 1  /* Peterson's mutex algorithm, two parallel processes 0 and 1 */
 2  bool flag[2] = false
 3  bool turn = 0
 4  byte count = 0
 5
 6  /* flag is initialized to all false, */
 7  /* and turn has the initial value 0  */
 8
 9  active [2] proctype peterson()
10  {
11      pid i = _pid; pid j = 1 - _pid
12      /* Infinite loop */
13  again:
14      /* [noncritical section] */
15      flag[i] = true
16      /* [trying section] */
17      turn = i
18      (flag[j] == false || turn != i)
19      count++
20      assert(count == 1)
21      /* [critical section] */
22      count--
23      flag[i] = false
24      goto again
25  }
```


Simulation mode

```
$ spin -u1000 peterson.pml
-------------
depth-limit (-u1000 steps) reached
#processes: 2
                flag[0] = 1
                flag[1] = 1
                turn = 1
                count = 0
1000:   proc  1 (peterson:1) peterson.pml:18 (state 3)
1000:   proc  0 (peterson:1) peterson.pml:23 (state 7)
2 processes created
```

Verification mode

```
$ spin -run peterson.pml

(Spin Version 6.4.8 -- 2 March 2018)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 28 byte, depth reached 22, errors: 0
...
```


Section 4.5

Semaphores

Atomic sequences of statements can be used to model synchronization primitives such a semaphores. The most widely known construct for synchronizing concurrent programs is the ***semaphore***. Here is a simple definition of a semaphore using concepts of Promela:

A semaphore `sem` is a variable of type **`byte`** (nonnegative integers). There are two *atomic* operations defined for a semaphore:

- `wait(sem)`: The operation is executable when the value of `sem` is positive; executing the operation decrements the value of `sem`.
- `signal(sem)`: The operation is always executable; executing the operation increments the value of `sem`.

Listing 4.6. The critical section problem with semaphores (sem_v6.pml)

```
byte sem = 1

active proctype P() {
  do
  :: printf("Noncritical section P\n")
     atomic {    /* wait(sem) */
       sem > 0
       sem--
     }
     printf("Critical section P\n")
     sem++       /* signal(sem) */
  od
}

active proctype Q() {
  do
  :: printf("Noncritical section Q\n")
     atomic {    /* wait(sem) */
       sem > 0
       sem--
     }
     printf("Critical section Q\n")
     sem++       /* signal(sem) */
  od
}
```

Advanced: Fairness of semaphores

The subject of semaphores is more complex than this simple example indicates. The difficulties arise when we try to define the *fairness* of the semaphore operations. Even when a verification is performed with weak fairness enabled (see Section 5.5), a computation for starvation is found because process Q can enter its critical section repeatedly while P does not.

In this computation, the only process that executes is process Q, which repeatedly executes its entire loop from line 16 to 24. The computation is weakly fair because P is enabled infinitely often (after Q executes `sem++` at line 23); the nonfair computation simply chooses not to execute the atomic statement from P when it is enabled.

The `signal` operation is usually defined to unblock one of the processes blocked on the semaphore (if any) as part of its atomic operation. A *strong semaphore* implements the set of blocked processes as a FIFO (first in-first out) queue; this is easy to model in Promela using channels. The `signal` operation of a *weak semaphore* unblocks an arbitrary element of the set; weak semaphores are harder to model in Promela.

Section 4.6

Nondeterminism in models of concurrent systems

Consider, for example, a communications system that must be able to receive and process an *arbitrary* stream of messages of several different types. A natural approach to modeling this requirement is to generate the messages stream by using a *random number* generator. If there are n message types m_0, m_1, ..., m_{n-1}, each message in the stream is obtained by generating a random number in the range 0 to $n - 1$.

However, this approach is flawed. While a random number generator can be used to obtain a random computation (and this is precisely what Spin does in random simulation mode), for verification *all* computations must be checked, not just those that happen to be generated randomly.

By design, Spin does not contain constructs for modeling probability or for specifying that an event must occur with a certain probability. The intended use of model checking is to detect errors that occur under complex scenarios that are unlikely to be discovered during system testing. "in a well-designed system, erroneous behavior should be impossible, not just improbable" [*The Spin Model Checker*, p.454].

In Spin, nondeterminism is used to model arbitrary values of data: whenever a value – such as a message type in a stream – is needed, a nondeterministic choice is made among all values in the range.

Subsection 4.6.1

Generating values nondeterministically

Suppose that we want to model a client-server system in which the client *nondeterministically* chooses which request to make; we can use an **if**-statement whose guards are always true:

```
active proctype Client() {
  if
  :: true -> request = 1
  :: true -> request = 2
  fi
  /* Wait for service */
  if
  :: true -> request = 1
  :: true -> request = 2
  fi
  /* Wait for service */
}
```

The code can be shortened by doing away with the expressions **true** which serve no purpose. Instead, the assignment statements themselves – which are always executable – can be used as guards:

```
active proctype Client() {
  if
  :: request = 1
  :: request = 2
  fi
  /* Wait for service */
  if
  :: request = 1
  :: request = 2
  fi
  /* Wait for service */
}
```


Of course, it doesn't make sense to model a client that generates only two requests. A **do**-statement can be used to model a client that generates an unending stream of requests in an arbitrary order:

```
active proctype Client() {
  do
  :: request = 1
  /* Wait for service */
  :: request = 2
  /* Wait for service */
  od
}
```

Subsection 4.6.2

Generating from an arbitrary range

We have shown how to model nondeterministically generated values from a small range:

```
byte number

if
:: number = 1
:: number = 2
:: number = 3
:: number = 4
fi
```

As the range of values gets larger, it becomes inconvenient to write alternatives for each values. The following Promela code shows how to choose nondeterministically values from an arbitrary range, in this case from 0 to 9:

```
#define LOW 0
#define HIGH 9

byte number = LOW

do
:: number < HIGH -> number++
:: break
od
```


As long as the value of `number` is less than `HIGH`, both alternatives are executable and Spin can choose either one. If it chooses the first one, the value of `number` is incremented; if it chooses the second, the loop is left and the current value of `number` is used in the subsequent code. It follows that the final value of `number` can be any value within the range.

Do not put any faith in the uniformity of the probability distribution of the "random numbers" generated using this technique. Assuming that Spin chooses uniformly between alternatives in the **do**-statement, the first value 0 has a probability of ½ while the last value 9 has a probability of 2^{-10}. Nondeterminism is used to generate arbitrary computations for verification, not random numbers for a faithful simulation.

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

NAME

select - non-deterministic value selection.

SYNTAX

```
select '(' varref ':' expr '..' expr ')'
```

DESCRIPTION

The `select` statement was added in Spin Version 6 to simplify writing a standard selection of a non-deterministic value from a specified range of values.

...

`Select` statements are internally converted into the corresponding Promela code, with the first statement issued being an assignment statement. This means that `select` statements are always executable (the guard statement is an assignment), but can take several steps to execute. More precisely, if there are `N` values in the range to choose from, then the `select` statement can take between 1 and `N` steps to arrive at the non-deterministically chosen value. The sequence of steps can be embedded in an `atomic` (but *not* in a `d_step`) to optimize the execution.

...

...

Caution 1: Because the `select` is implemented with a non-deterministic do-loop (making it possible to use expressions for the ranges that are evaluated at run-time), you will not get very random behavior in *simulation runs*. Note that each time through the loop, a random simulation will give equal odds to selecting the end of the loop or its continuation, until the upper-limit is reached.

That means that values close to the start are much more likely to be picked in simulation runs than values close to the end. The behavior in verification runs is of course guaranteed to be correct, with all possible choices being verified.

...

...

EXAMPLES

A simple example of the use of a `select` statement is as follows,

```
select (i : 8..17)
assert(i >= 8 && i <= 17)
```

which is expanded into the following Promela code fragment that can non-deterministically select any value in the range provided:

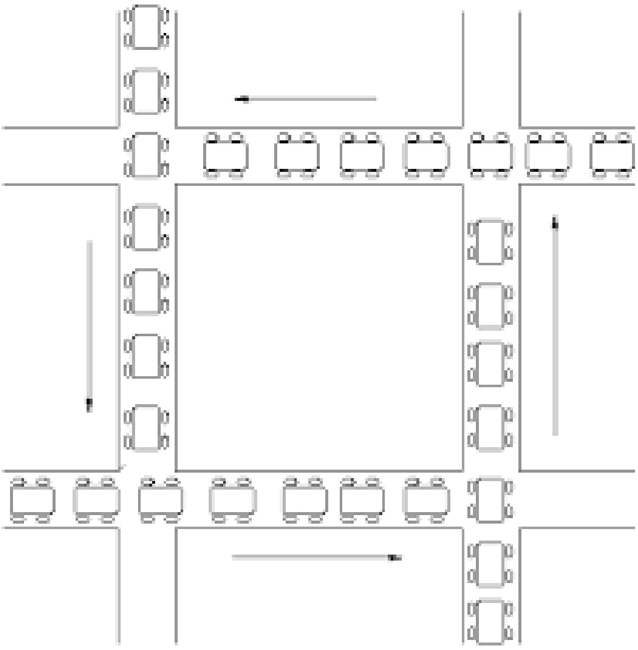
```
i = 8
do
:: i < 17 -> i++
:: break
od
```

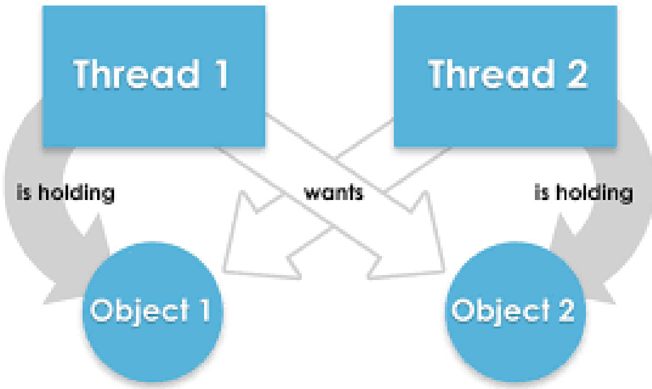
Section 4.7

Termination of processes

Subsection 4.7.1

Deadlock




Thread 1
Thread 2
is holding
wants
is holding
Object 1
Object 2

Unfortunately, the program in Listing 4.2 (diap. 37) is not a correct solution of the critical section problem. The processes of the program consist of loops with no **`goto`** or **`break`** statements, so the program should never terminate. If you run several random simulations of the program, you will likely encounter a computation in which execution terminates with the output `timeout`. This means that *no* statements are executable, a condition called *deadlock*.

It is quite easy to construct the computation that leads to deadlock. Simply execute statements in perfect interleaving (one statement alternately from each process); both `wantP` and `wantQ` are set to true (lines 9, 20) and then both processes are blocked waiting for the other one to set its variable to false (lines 10, 21).

An attempt at verification will discover an error called an *invalid end state*:

```
pan: invalid end state (at depth 8)
```

By default, a process that does terminate must do so after executing its last instruction, otherwise it is said to be in an invalid end state. This error is checked for regardless of any other correctness specifications. This default behavior can be overridden as described in the next subsection.

Simulation mode

```
$ spin third-deadlock_v6.pml
          Non critical section Q
      Non critical section P
      timeout
#processes: 2
                wantP = 1
                wantQ = 1
  4:    proc  1 (Q:1) third-deadlock_v6.pml:21 (state 3)
  4:    proc  0 (P:1) third-deadlock_v6.pml:10 (state 3)
2 processes created
```


Verification mode

```
$ spin -run third-deadlock_v6.pml
pan:1: invalid end state (at depth 8)
pan: wrote third-deadlock_v6.pml.trail

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 20 byte, depth reached 9, errors: 1
...
```

Subsection 4.7.2

End states

Listing 4.7. A client-server program with end states (`end.pml`)

```
byte request = 0

active proctype Server1() {
  do
  :: request == 1 ->
       printf("Service 1\n")
       request = 0
  od
}

active proctype Server2() {
  do
  :: request == 2 ->
       printf("Service 2\n")
       request = 0
  od
}

active proctype Client() {
  request = 1
  request == 0
  request = 2
  request == 0
}
```

This Promela program is a reasonable model of a very simple *client-server* system. However, if you simulate or verify it in Spin, you will receive an error message that there is an invalid end state. The reason is that while the client executes a finite number of statements and then terminates, the servers are always blocked at the guard of the **do**-statement waiting for it to become executable.

Now, this is acceptable behavior because servers should wait indefinitely and be ready to supply a service whenever it is needed. Since the server cannot know how many requests it will receive, it is unreasonable to require termination of a process modeling a server.

You can indicate that a control point within a process is to be considered a valid end point even though it is not the last statement of the process by prefixing it with a label that begins with end:

```
active proctype Server1() {
endserver:
  do
  :: request == 1 -> ...
  od
}
```

Subsection 4.7.3

The order of process termination

A process *terminates* when it has reached the end of its code, but it is considered to be an active process until it *dies*. Spin manages process allocation in the LIFO (last in-first out) order of a stack, so a process can die only if it is the most recent process that was created. Usually, the distinction between process termination and death is not an issue, but it can sometimes explain why a program does not end as expected.

Process termination and death are demonstrated by the program in Listing 4.8 (next slide). The two servers each perform one service and then terminate, incrementing the variable `finished` that counts the number of processes that have terminated.

Listing 4.8. Client-server termination (`end1.pml`)

```
byte request = 0
byte finished = 0

active proctype Server1() {
  request == 1
  request = 0
  finished++
}

active proctype Server2() {
  request == 2
  request = 0
  finished++
}

active proctype Client() {
  request = 1
  request == 0
  request = 2
  request == 0
  finished == 2
}
```


Since processes created by **`active proctype`** are instantiated in order written, the two server processes do not die until the client process finds `finished == 2` and terminates.

The output is just as we expect it to be:

```
11:    proc  2 (Client) terminates
11:    proc  1 (Server2) terminates
11:    proc  0 (Server1) terminates
```

Suppose now that we change line 21 to `finished == 3` so that the client process does not terminate. By the LIFO rule, the server processes will not terminate, and the simulation goes into a state called `timeout` in which no process is at an executable statement:

```
timeout
#processes: 3
2 Client               2         0
1 Server               2         0
0 Server               2         0
```

All three processes are still active, though none are executable.

Next, move the process `Client` so that it appears *before* the server processes in the source code. Now, the server processes are created after the client process so they can terminate without waiting for the client process, which is blocked, hopelessly waiting for `finished` to receive the value 3:

```
timeout
#processes: 1
0 Client               2         0
```