

Coursework 1: PROMELA and SPIN

Overview

The assignment asks to write three PROMELA specifications that verify mutual exclusion and process fairness. I have completed all three parts. In addition, I have written several specifications for the first task, 2 specifications for the second. I have also looked into one of the points of part 4.

Part 1

The first task is about mutual exclusion between three processes that each have a ‘critical section’. No two processes should be in their critical sections at the same time. I have written four PROMELA specifications.

First, the one that satisfies the assignment to the letter is ‘part1_simple.pml’. It runs three different processes that each wait for their turn to enter their critical sections. An invariant ensures that no two processes are in their critical sections at the same time.

Second, ‘part1_simple_infinite.pml’ is the same, except it lets the processes run forever. When a process leaves its critical section, it goes back to waiting for its turn.

Third, ‘part1_array.pml’ defines an array of processes, rather than three separate ones. The array is of size 4. The process with index 0 terminates immediately. Thus, the three remaining processes are indexed 1, 2, and 3, as required by the assignment. Again, they each wait for their turn to enter their critical sections, and an invariant ensures no two processes enter their critical sections at the same time.

Finally, ‘part1_array_infinite.pml’ is the same as ‘part1_array.pml’, except the processes run forever. Process 0 terminates early, but processes 1, 2, and 3 return to waiting their turns after leaving their critical sections.

Both the ‘simple’ and the ‘array’ variations satisfy the mutual exclusion property.

The ‘simple’ variant does that by using PROMELA’s blocking on boolean statements. A process blocks until its boolean condition is met. The variable ‘x’ determines which process can have a go next, so only one of the conditions can be true at the same time. After a process passes the condition, it first changes its state variable (s_i) to true and then changes the value of ‘x’. Thus, all conditions become false (because one of the states is true). Changing the turn after that does not allow a process to enter its critical section. At the end of its critical section, a process changes its state back to false. Now all states are false, and ‘x’ determines which process can go next. The invariant ‘Checker’ has three asserts that are run in an infinite loop. They check that no two states are true at the same time. Running verification in Spin ensures that all asserts pass in all states of the program. Spin verification goes through each possible trace of execution. The asserts are put inside an atomic block, so that they are always executed together and thus reduce the search states. Here is the output produced by iSpin:

The screenshot shows the iSpin Version 6.4.8 interface. The left pane displays a PROMELA model with three parallel processes (P1, P2, P3) and a checker. The middle pane shows the configuration: Safety (invalid endstates, assertion violations, assertions), Storage Mode (exhaustive), and Search Mode (depth-first search, partial order reduction). The right pane shows the results of the verification run, indicating that no errors or assertion violations were found.

```

1  bool s1, s2, s3;
2  mtype = {one,two,three};
3  mtype x = one;
4
5  active proctype P1() {
6    !s1 && !s2 && !s3 && x == one;
7    s1 = true;
8    x = two;
9    s1 = false;
10 }
11
12 active proctype P2() {
13   !s1 && !s2 && !s3 && x == two;
14   s2 = true;
15   x = three;
16   s2 = false;
17 }
18 active proctype P3() {
19   !s1 && !s2 && !s3 && x == three;
20   s3 = true;
21   x = one;
22   s3 = false;
23 }
24
25 active proctype Checker() {
26   do
27     :: atomic {
28       assert(!s1 || !s2);
29       assert(!s2 || !s3);
30       assert(!s3 || !s1);
31     }
32   od
33 }

```

Results summary:

- Full statespace search for: never claim, assertion violations, cycle checks, invalid end states
- State-vector 36 byte, depth reached 12, errors: 0
- 13 states, stored
- 13 states, matched
- 26 transitions (= stored+matched)
- 0 atomic steps
- hash conflicts: 0 (resolved)
- Stats on memory usage (in Megabytes):
 - 0.901 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 P1 (0 of 5 states)
- unreached in proctype P2 (0 of 5 states)
- unreached in proctype P3 (0 of 5 states)
- unreached in proctype Checker part1_simple.pml:33, state 8, ~end~ (1 of 8 states)
- pan: elapsed time 0 seconds
- No errors found -- did you verify all claims?

As can be seen, no errors are found, no assertions are violated.

The ‘array’ variant also uses PROMELA’s blocking on boolean statements to satisfy the mutual exclusion property. A process blocks until it is its turn to enter the critical section. After that it waits until all states are false. In the critical section, a process first changes the state, and then changes the value of ‘x’. Again, ‘x’ ensures that no two processes are waiting for the states to be false. At the end of its critical section, a process sets its state to false. In the ‘array’ variant, the ‘Checker’ invariant loops through all pairs of states, ensuring no two states are true at the same time. This is equivalent to having only one true state. Similar to the approach in ‘simple’, this PROMELA specification goes through all asserts in an atomic block. This way the search space for Spin’s exhaustive search is reduced. Here is the output produced by iSpin:

The screenshot shows the iSpin Version 6.4.8 interface. The left pane displays a PROMELA model for an array variant with N processes. The middle pane shows the configuration: Safety (invalid endstates, assertion violations, assertions), Storage Mode (exhaustive), and Search Mode (depth-first search, partial order reduction). The right pane shows the results of the verification run, indicating that no errors or assertion violations were found.

```

1  #define N 4
2  bool s[N];
3  byte x = 1;
4
5  active [N] proctype P() {
6    pid me = _pid;
7
8    // Only processes with indexes 1 to 3 should run.
9    if
10     :: me == 0 -> goto leave_critical;
11     :: else -> skip;
12   fi
13
14   // Wait until x says that it's my turn.
15   me == x;
16
17   // Now that it's my turn, wait until all s are false.
18   // Note that only one instance will be here at any one time.
19   byte i;
20   wait_release:
21   for (i := 1; i < (N-1); i++) {
22     if
23     :: s[i] -> goto wait_release;
24     :: else -> skip;
25   fi
26 }
27
28 // Critical section.
29 s[me] = true;
30 if
31 :: x != N -> x = 1;
32 :: else -> x = x+1;
33 fi
34
35 leave_critical:
36 s[me] = false;
37 }
38
39 active proctype Checker() {
40   int i, j;
41   do
42     :: atomic {
43       for (i := 0; i < (N-1); i++) {
44         for (j := i+1; j < (N-1); j++) {
45           if
46           :: i != j -> assert(!s[i] || !s[j]);
47           :: else -> skip;

```

Results summary:

- Full statespace search for: never claim, assertion violations, cycle checks, invalid end states
- State-vector 60 byte, depth reached 135, errors: 0
- 239 states, stored
- 197 states, matched
- 436 transitions (= stored+matched)
- 4524 atomic steps
- hash conflicts: 0 (resolved)
- Stats on memory usage (in Megabytes):
 - 0.020 equivalent memory usage for states (stored)(State-vector + overhead)
 - 0.297 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 (0 of 31 states)
- unreached in proctype Checker part1_array.pml:53, state 27, ~end~ (1 of 27 states)
- pan: elapsed time 0 seconds
- No errors found -- did you verify all claims?

The ‘infinite’ variants differ only in the labels that they use to return to the start of the process. The output produced by iSpin for them is the same.

Part 2

The second task is about mutual exclusion between an array of processes. No two processes should print at the same time. I have written two PROMELA specifications called ‘part2_lock.pml’ and ‘part2_turn.pml’.

In both of these, an invariant ensures that no two processes are printing at the same time. The invariant is almost the same as the one from the ‘array’ variant of part 1. The only difference is the iterated array. Again, an atomic block reduces the states for Spin’s exhaustive search verification. The verification will pass even without the atomic block, as Spin will exhaust the cases where the full ‘Checker’ is run after each statement of the other processes.

The ‘turn’ variant is similar to how part 1 utilized a variable to keep track of which process can enter the critical section. In this PROMELA specification, ‘turn’ is an integer that can take a random value between 0 and N (the number of processes that want to print). The value is set using PROMELA’s non-determinism in do-loops. The loop that changes ‘turn’ will increment it and decrement it a random number of times before moving on. A boolean variable ‘turnSet’ is used to determine whether or not ‘turn’ is currently being changed. The value of ‘turn’ is set once in the init process, and then every time a process finishes printing.

A process sets its respective ‘print’ value to true before printing, and then to false after printing but before changing the value of ‘turn’.

The processes that wish to print wait for their turn via PROMELA’s blocking on boolean statements. A process blocks until ‘turnSet’ is true and the value of turn is its id. This way no two processes can be printing at the same time, satisfying the mutual exclusion property. Here is the output from iSpin:

The screenshot displays the iSpin Version 6.4.8 interface. The left pane shows the PROMELA code for 'part2_turn.pml'. The right pane shows the search results for a full statespace search.

PROMELA Code (part2_turn.pml):

```

1 #define N 100
2 bool print[N];
3 bool turnSet = false;
4 int turn = N/2;
5
6 active (N) prototype P() {
7   pid me = _pid;
8
9   wait turn;
10  turnSet && me == turn;
11
12  // State printing
13  print[me] = true;
14  print("Now P%d can print\n", me);
15  print[me] = false;
16  turnSet = false;
17  do
18    :: turn < (N-1) -> turn++;
19    :: turn > 0 -> turn--;
20    :: 0 <= turn && turn < N -> break;
21  od
22  turnSet = true;
23
24  // Repeat infinitely often.
25  goto wait_turn;
26
27
28 active prototype Checker() {
29   int i, j;
30   do
31     :: atomic {
32       for (i: 0 .. (N-1)) {
33         for (j: i .. (N-1)) {
34           if
35             :: i == j -> assert(!print[i] || !print[j]);
36             else -> skip;
37           fi
38         }
39       }
40     }
41   od
42
43   init {
44     do
45       :: turn++;
46       :: turn--;
47     od
48   }
49 }

```

Search Results:

```

spin -a part2_turn.pml
gcc -DMEMLIM=1024 -O2 -DSAFETY -DNOCLAIM -w -o pan pan.c
pan -m10000
Pid: 23666
error: max search depth too small
(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 932 byte, depth reached 9999, errors: 0
51620 states, stored
86950 states, matched
138570 transitions (= stored+matched)
22423626+98 atomic steps
hash conflicts: 12 (resolved)

Stats on memory usage (in Megabytes):
  47.260 equivalent memory usage for states (stored*(State-vector + overhead))
  22.412 actual memory usage for states (compression: 47.42%)
  state-vector as stored = 427 bytes + 23 byte overhead
  128,000 memory used for hash table (-w24)
  0.534 memory used for DFS stack (-m10000)
  150,800 total actual memory usage

unreached in prototype P
  part2_turn.pml:26, state 17, "end-"
  (1 of 17 states)
unreached in prototype Checker
  part2_turn.pml:42, state 27, "end-"
  (1 of 27 states)
unreached in init
  (0 of 9 states)

pan: elapsed time 6.04 seconds
No errors found -- did you verify all claims?

```

The ‘lock’ variant uses an integer variable called ‘lock’ to prevent more than one process from printing. When a process wishes to print, it increments the value of ‘lock’ by one. If it is the only process that has done that, it can then print. Similar to the ‘turn’ variant, a process will first set its respective value in the ‘print’ array to true, then print, and then set that value back to false. After that it will release the ‘lock’.

Race conditions (where multiple processes increment the ‘lock’ at the same time), are handled using a simple back-off procedure. If more than one process is holding the ‘lock’, then decrement the ‘lock’ and go back up. Spin can support up to 256 processes. This means that ‘lock’ will never overflow (the maximum integer value is $2^{31}-1$). The ‘lock’ ensures that no two process can print at the same time and that is verified by the invariant. Here is the output from iSpin:

The screenshot shows the Spin Version 6.4.8 interface. The left pane contains the Spin source code for a mutual exclusion algorithm using a lock. The code defines a process P0 that increments a lock variable to enter a critical section, prints, and then releases the lock. A checker process is also defined to verify the invariant. The right pane displays the search results, showing that the search was not completed due to memory constraints and a large state space.

```

Spin Version 6.4.8 -- 2 March 2018 :: iSpin Version 1.1.4 -- 27 November 2014

Safety
  * safety
    + invalid endstates (deadlock)
    + assertion violations
    + x/rxs assertions
  * non-progress cycles
  * acceptance cycles
  * enforce weak fairness constraint

Storage Mode
  * exhaustive
    + minimized automata (slow)
    + collapse compression
  * hash-compact
  * bitstate/supertrace
  * do not use a never claim or ltl property
  * do 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

Show Error Tracing Options
Show Advanced Parameter Settings

Run Stop

pan -m10000
Pid: 27129
error: max search depth too small
Depth= 8999 States= 1e+08 Transitions= 556.952 t= 6.82 R= 2e+05
Depth= 8999 States= 2e+06 Transitions= 2.01e+08 Memory= 985.272 t= 12.1 R= 2e+05
pan: reached -DMEMLIM bound
1.07395e+09 bytes used
102400 bytes more needed
hint: to reduce memory, recompile with
1.07374e+09 bytes limit
-DCOLLAPSE # good, fast compression, or
-DMA-324 # better/slower compression, or
-DHC # hash-compaction, approximation
-DBITSTATE # supertrace, approximation
(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
+ Partial Order Reduction

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

State-vector 924 byte, depth reached 9999, errors: 0
2090304 states, stored
2.0804596e+08 states, matched
2.1015596e+08 transitions (= stored+matched)
52126473 atomic steps
hash conflicts: 131262 (resolved)

Stats on memory usage (in Megabytes):
1897.783 equivalent memory usage for states (stored*(State-vector + overhead))
897.741 actual memory usage for states (compression: 47.30%)
state-vector as stored = 422 byte + 28 byte overhead
128.000 memory used for hash table (=24)
0.534 memory used for DFS stack (=m10000)
2.331 memory lost to fragmentation
1063.944 total actual memory usage

pan: elapsed time 12.5 seconds
No errors found -- did you verify all claims?

```

Due to the race condition, there are a lot more possible states for Spin to explore. The output above shows how Spin increased its memory usage and maximum depth for the iterations.

Part 3

The third task is about Linear Time Logic (LTL) and reflecting on when the formulae hold. The PROMELA specification has three processes. Two increment the value of ‘x’ and one sets the value of ‘y’ to that of ‘x’. Spin supports weak fairness. In this PROMELA specification, all three processes are enabled at all times. Thus, it follows that each process will get its turn infinitely often.

The first statement is “x is always odd”. In LTL, it is written as “ $\Box (x \% 2 == 1)$ ”. This statement holds only when P2 never gets to run. That is the case of no fairness. Spin supports weak fairness, therefore P2 is guaranteed to run infinitely often from a certain time onwards. When it executes, it will increment ‘x’ by 1 and thereby change its parity. So, this statement does not hold in weak fairness. Here’s the result of running Spin with this LTL:

Spin Version 6.4.8 – 2 March 2018 :: Spin Version 1.1.4 – 27 November 2014

Safety

- ☐ safety
- ☒ + invalid endstates (deadlock)
- ☒ + assertion violations
- ☒ + x/r/s 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

Run Stop

```

1 #define X_ODD (x%2 == 1)
2 byte x=1;
3 byte y=0;
4
5 // a { always X_ODD }
6 // Holds when P2 does not get to execute at all. No fairness.
7
8 // b { ! (eventually (always X_ODD)) }
9 // Negate and Spin check for reverse condition.
10 // Spin succeeds, so the original condition is false.
11
12 // c { ! (eventually (always (eventually X_ODD))) }
13 // Weak fairness, P2 will eventually make x odd.
14 // No fairness, P2 never executes, P1 does not change x's parity, so x is always odd.
15
16 // d { always (y<=x) }
17 // Weak fairness, once x reaches 255, do not execute P1 or P2,
18 // as those will result in a byte overflow.
19 // No fairness, never execute P3, so y will always be 0.
20
21 // e { always ((y==x) implies (eventually (y!=x))) }
22 // Weak fairness, eventually P1 or P2 will run after P3
23 // No fairness, nothing runs after P3
24
25 active prototype P1() {
26   do
27     : x=x+2;
28   od
29 }
30
31 active prototype P2() {
32   do
33     : x=x+1;
34   od
35 }
36
37 active prototype P3() {
38   do
39     : y<x -> y=x;
40   od
41 }

```

spin -a part3.pml
 ltl a: ! ((x%2==1))
 go: -DMEMLM=1024 -O2 -w -o pan.pan.c
 -pan -m10000 -a
 Pid: 1817
 pan: assertion violated: ! ((x%2==1)) (at depth 6)
 pan: wrote part3.pml.trail

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

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

State-vector 44 byte, depth reached 6, errors: 1
 4 states, stored
 0 states, matched
 4 transitions (= stored+matched)
 0 atomic steps
 hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
 0.000 equivalent memory usage for states (stored) (State-vector + overhead)
 0.290 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

pan: elapsed time 0.01 seconds
 To replay the error-trail, goto Simulate/Replay and select "Run"

The second statement is “It is possible that from a certain point onwards x is always odd”. The way to check statements of the type “It is possible that A” is to negate A and verify that $\neg A$ fails. If Spin finds a case where $\neg A$ is false, then in that case A is true and thus the statement is true. If $\neg A$ is true in all executions, then A is never possible. In our statement, A is “from a certain point onwards x is always odd”. In LTL, it is written as “ $\Diamond \Box (x \% 2 == 1)$ ”. Given weak fairness, P2 will eventually get a turn and thus change the parity of ‘x’. So, the parity of ‘x’ will never remain constant. Similar to the first statement, this one would be true if P2 does not get its turn (no fairness). Spin fails to verify the negation of A because it is not possible to simulate infinite execution. It simulates the case where

Spin Version 6.4.8 – 2 March 2018 :: Spin Version 1.1.4 – 27 November 2014

Safety

- ☐ safety
- ☒ + invalid endstates (deadlock)
- ☒ + assertion violations
- ☒ + x/r/s 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

Run Stop

```

1 #define X_ODD (x%2 == 1)
2 byte x=1;
3 byte y=0;
4
5 // a { always X_ODD }
6 // Holds when P2 does not get to execute at all. No fairness.
7
8 // b { ! (eventually (always X_ODD)) }
9 // Negate and Spin check for reverse condition.
10 // Spin succeeds, so the original condition is false.
11
12 // c { ! (eventually (always (eventually X_ODD))) }
13 // Weak fairness, P2 will eventually make x odd.
14 // No fairness, P2 never executes, P1 does not change x's parity, so x is always odd.
15
16 // d { always (y<=x) }
17 // Weak fairness, once x reaches 255, do not execute P1 or P2,
18 // as those will result in a byte overflow.
19 // No fairness, never execute P3, so y will always be 0.
20
21 // e { always ((y==x) implies (eventually (y!=x))) }
22 // Weak fairness, eventually P1 or P2 will run after P3
23 // No fairness, nothing runs after P3
24
25 active prototype P1() {
26   do
27     : x=x+2;
28   od
29 }
30
31 active prototype P2() {
32   do
33     : x=x+1;
34   od
35 }
36
37 active prototype P3() {
38   do
39     : y<x -> y=x;
40   od
41 }

```

spin -a part3.pml
 ltl b: !<= (! ((x%2==1)))
 go: -DMEMLM=1024 -O2 -w -o pan.pan.c
 -pan -m10000 -a
 Pid: 1946
 pan: acceptance cycle (at depth 766)
 pan: wrote part3.pml.trail

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

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

State-vector 44 byte, depth reached 1022, errors: 1
 766 states, stored
 0 states, matched
 766 transitions (= stored+matched)
 0 atomic steps
 hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
 0.063 equivalent memory usage for states (stored) (State-vector + overhead)
 0.290 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

pan: elapsed time 0.01 seconds
 To replay the error-trail, goto Simulate/Replay and select "Run"

only P1 is executed (Spin trail and trail execution included in submission). Spin output above.

The third statement is “It is possible that from a certain point onwards x is indefinitely often odd”. Similar to the second statement, we need to verify the negation of A . In this case A is “from a certain point onwards x is indefinitely often odd”. In LTL, it is written as “ $\Diamond \Box \Diamond (x \% 2 == 1)$ ”. Spin fails to verify the negation of A . This means that A is possible. Given weak fairness, $P2$ will eventually get a turn and thus change the parity of ‘ x ’. This will happen infinitely often as $P2$ is executed infinitely often. Thus ‘ x ’ will always eventually be odd. With no fairness, $P2$ will not get its turn and so the statement will be true if and only if ‘ x ’ is odd after $P2$ ’s last execution. Here’s the result of running Spin with this LTL:

The screenshot shows the Spin 6.4.8 interface. On the left, the LTL formula is defined as follows:

```

1 #define X_ODD (x%2 == 1)
2 byte x=1;
3 byte y=0;
4
5 //Itl a ( always X_ODD )
6 // Holds when P2 does not get to execute at all. No fairness.
7
8 //Itl b ( !eventually (always X_ODD) )
9 // Negate and Spin check for reverse condition.
10 // Spin succeeds, so the original condition is false.
11
12 //Itl c ( !eventually (always (eventually X_ODD)) )
13 // Weak fairness, P2 will eventually make x odd.
14 // No fairness, P2 never executes, P1 does not change x's parity, so x is always odd.
15
16 //Itl d ( always (x<=x) )
17 // Weak fairness, once x reaches 255, do not execute P1 or P2,
18 // as those will result in a byte overflow.
19 // No fairness, never execute P3, so y will always be 0.
20
21 //Itl e ( always ((y==x) implies (eventually (y!=x))) )
22 // Weak fairness, eventually P1 or P2 will run after P3
23 // No fairness, nothing runs after P3
24
25 active prototype P1() {
26   do
27     : x=x+2;
28   od
29 }
30
31 active prototype P2() {
32   do
33     : x=x+1;
34   od
35 }
36
37 active prototype P3() {
38   do
39     : y<x -> y=x;
40   od
41 }

```

On the right, the search results are displayed:

```

spin -a pan3.sml
Itl c: 1 (<> (! (<> (! ((x%2==1))))))
gcc -DMEMLIM=1024 -O2 -w -o pan pan.c
pan -m10000 -a
Pid: 2097
pan:1: acceptance cycle (at depth 1528)
pan: wrote pan15.pml.trail
(Spin Version 6.4.8 - 2 March 2018)
Warning: Search not completed
+ Partial Order Reduction

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

State-vector 44 byte, depth reached 2295, errors: 1
1148 states, stored (1149 visited)
130 states, matched
1279 transitions (= visited+matched)
0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.079 equivalent memory usage for states (stored*(State-vector + overhead))
0.289 actual memory usage for states
128.000 memory used for hash table (-w24)
0.534 memory used for DFS stack (-m10000)
128.750 total actual memory usage

pan: elapsed time 0.01 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"

```

The fourth statement is “It is always true that $y \leq x$ ”. In LTL, it is written as “ $\Box (y \leq x)$ ”. This statement does not hold in weak fairness. The variable ‘ x ’ is of type byte. The largest value it can have is 255. Processes $P1$ and $P2$ can both attempt to increase ‘ x ’ beyond that which will cause it to overflow to 0. In that case, the LTL can only hold if ‘ y ’ is never changed. The value of ‘ y ’ gets changed in $P3$. So, by weak fairness there will be a case where ‘ y ’ is set to a value greater than 0 and then ‘ x ’ will overflow to 0. With no fairness, $P3$ will never get a turn, and ‘ y ’ will remain 0, and then the property holds. Here’s the result of running Spin with this LTL:

Spin Version 6.4.8 - 2 March 2018 :: Spin Version 1.1.4 - 27 November 2014

Safety

- ☒ safety
- ☒ + invalid endstates (deadlock)
- ☒ + assertion violations
- ☒ + x/r/x assertions

Storage Mode

- ☒ exhaustive
- ☐ + minimized automata (slow)
- ☐ + collapse compression
- ☐ hash-compact ☐ bitstate/supertrace

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

Never Claims

- ☐ do not use a never claim or ltl property
- ☒ use claim
- claim name (opt):

Run **Stop** **Save Result in:** **pan.out**

```

1 #define X_ODD (x%2 == 1)
2 byte x=1;
3 byte y=0;
4
5 //ltl a { always X_ODD }
6 // Holds when P2 does not get to execute at all. No fairness.
7
8 //ltl b { (eventually (always X_ODD)) }
9 // Negate and Spin check for reverse condition.
10 // Spin succeeds, so the original condition is false.
11
12 //ltl c { (eventually (always (eventually X_ODD))) }
13 // Weak fairness, P2 will eventually make x odd.
14 // No fairness, P2 never executes, P1 does not change x's parity, so x is always odd.
15
16 ltl d { always (y<x) }
17 // Weak fairness, once x reaches 255, do not execute P1 or P2,
18 // as those will result in a byte overflow.
19 // No fairness, never execute P3, so y will always be 0.
20
21 //ltl e { always ((y==x) implies (eventually (y!=x))) }
22 // Weak fairness, eventually P1 or P2 will run after P3
23 // No fairness, nothing runs after P3
24
25 active proctype P1 {
26   do
27     :: x==x+2;
28   od
29 }
30
31 active proctype P2 {
32   do
33     :: x=x+1;
34   od
35 }
36
37 active proctype P3 {
38   do
39     :: y<x -> y=x;
40   od
41 }

```

Spin -a pan3.pml
ltd: 0 ((y<x))
gcc-DMEMLIM=1024 -O2 -w -o pan.pan.c
pan -m10000 -a
Pid: 2268
pan: assertion violated: 1 ((y<x)) (at depth 1530)
pan: wrote pan3.pml.trail
(Spin Version 6.4.8 - 2 March 2018)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:

- never claim
- assertion violations + (if within scope of claim)
- acceptance cycles + (fairness disabled)
- invalid end states - (disabled by never claim)

State-vector 44 byte, depth reached 1530, errors: 1
766 states, stored
0 states, matched
766 transitions - stored+matched
0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.053 equivalent memory usage for states (stored+State-vector + overhead)
0.290 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

pan: elapsed time 0.09 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"

The fifth statement is “It is always true that when $y = x$ it follows that at some point $y \neq x$.” In LTL, it is written as $\Box ((y=x) \rightarrow (\Diamond y \neq x))$. This statement holds in weak fairness. The process P3 sets the value of ‘y’ to that of ‘x’. Process P1 and P2 change the value of ‘x’ without affecting that of ‘y’. This will happen eventually, as weak fairness guarantees that P1 and P2 will get their turns. With no fairness, P1 and P2 will never get a turn, and then the property does not hold. Here’s the result of running Spin with this LTL:

Spin Version 6.4.8 - 2 March 2018 :: Spin Version 1.1.4 - 27 November 2014

Safety

- ☒ safety
- ☒ + invalid endstates (deadlock)
- ☒ + assertion violations
- ☒ + x/r/x assertions

Storage Mode

- ☒ exhaustive
- ☐ + minimized automata (slow)
- ☐ + collapse compression
- ☐ hash-compact ☐ bitstate/supertrace

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

Never Claims

- ☐ do not use a never claim or ltl property
- ☒ use claim
- claim name (opt):

Run **Stop** **Save Result in:** **pan.out**

```

1 #define X_ODD (x%2 == 1)
2 byte x=1;
3 byte y=0;
4
5 //ltl a { always X_ODD }
6 // Holds when P2 does not get to execute at all. No fairness.
7
8 //ltl b { (eventually (always X_ODD)) }
9 // Negate and Spin check for reverse condition.
10 // Spin succeeds, so the original condition is false.
11
12 //ltl c { (eventually (always (eventually X_ODD))) }
13 // Weak fairness, P2 will eventually make x odd.
14 // No fairness, P2 never executes, P1 does not change x's parity, so x is always odd.
15
16 //ltl d { always (y<x) }
17 // Weak fairness, once x reaches 255, do not execute P1 or P2,
18 // as those will result in a byte overflow.
19 // No fairness, never execute P3, so y will always be 0.
20
21 //ltl e { always ((y==x) implies (eventually (y!=x))) }
22 // Weak fairness, eventually P1 or P2 will run after P3
23 // No fairness, nothing runs after P3
24
25 active proctype P1 {
26   do
27     :: x==x+2;
28   od
29 }
30
31 active proctype P2 {
32   do
33     :: x=x+1;
34   od
35 }
36
37 active proctype P3 {
38   do
39     :: y<x -> y=x;
40   od
41 }

```

ltd: 1 ((y==x)) ((y!=x))
gcc-DMEMLIM=1024 -O2 -w -o pan.pan.c
pan -m10000 -a
Pid: 2434
(Spin Version 6.4.8 - 2 March 2018)
+ Partial Order Reduction

Full statespace search for:

- never claim
- assertion violations + (if within scope of claim)
- acceptance cycles + (fairness disabled)
- invalid end states - (disabled by never claim)

State-vector 44 byte, depth reached 2046, errors: 0
132093 states, stored (132858 visited)
229247 states, matched
362105 transitions - visited+matched
0 atomic steps
hash conflicts: 315 (resolved)

Stats on memory usage (in Megabytes):
9.020 equivalent memory usage for states (stored+State-vector + overhead)
5.270 actual memory usage for states (compression: 58.11%)
state-vector as stored - 14 byte + 28 byte overhead
128.000 memory used for hash table (-w24)
0.534 memory used for DFS stack (-m10000)
133.710 total actual memory usage

unreached in proctype P1
pan3.pml:29, state 5, "end"
(1 of 5 states)

unreached in proctype P2
pan3.pml:34, state 5, "end"
(1 of 5 states)

unreached in proctype P3
pan3.pml:39, state 6, "end"
(1 of 6 states)

unreached in claim e
spin_rvr.tmp:10, state 13, "end"
(1 of 13 states)

pan: elapsed time 0.07 seconds
No errors found - did you verify all claims?

Part 4

The fourth task is about detailing different aspects of Spin as a verification tool. I have focused on its current limitations. Several sources^[1, 2, 3] have pointed out a couple of limitations that apply to most software verification tools, including Spin. The first one is “state space explosion”. Real systems often have unlimited number of states and huge ranges for variables’ values. In PROMELA, processes can be interleaved in any way, and Spin stores explicit state. It would, therefore, need infinite memory to explore the state space of an infinite system. The study [1] recommends looking into other formal verification methods, whereas [2] suggests simplifying the model. The latter leads to the second common problem of verification tools.

Creating an accurate PROMELA model of a real system is often impossible. Programs often have multiple variables of different types and PROMELA is limited to numbers of different ranges. Furthermore, there is no guarantee that a verified model will bring information about the real system^[2].

A limitation specific to Spin is lack of real-time capabilities. It is pointed out in [1] and [2], that Spin does not natively support checking repetition cycles between events. In [3], it is mentioned that modifications and extensions of Spin can achieve this.

A final, small, Spin-specific limitation is that number of simultaneously running processes is limited to 255^[4]. I have not been able to find out why the maximum is not 256. If we use 8 bits for process ids, we have a total of 256 possible ids. Process id 0 is not taken by a system process because creating 10 processes assigns ids from 0 to 9. I suppose the id 255 is used for a process that manages the others but I have not been able to find any confirmation.

On top of these limitations, I have also read a bit about how Spin handles LTL formulae. It first transforms them into never claims. Spin then represents the never claim as a Büchi automata^[5]. A Büchi automata accepts an input sequence if it visits at least one of the final states infinitely often. Such an automata is designed to take an infinite input sequence. Thus, a never claim succeeds if it reaches the end of its input or if it passes through a final (aka ‘acceptance’) state infinitely often.

Bibliography

[1] A Survey of Tools for Model Checking and Model-Based Development

Elisabeth A. Strunk, M. Anthony Aiello, John C. Knight, Eds.

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.436.1944&rep=rep1&type=pdf>

[2] Applications for the Spin Model Checker

Ville R. Koskinen, Juha Plosila

<http://www.tucs.fi/publications/attachment.php?fname=TR782.pdf>

Note: this link initiates a PDF download.

[3] The Model Checker Spin

Gerard J. Holzmann

<http://spinroot.com/spin/Doc/ieee97.pdf>

[4] Spin Manual

<http://spinroot.com/spin/Man/active.html>

[5] Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker

Diego LatellaIstvan MajzikMieke Massink

<https://link.springer.com/article/10.1007/s001659970003>