Concurrent Programming

Exercise Booklet 9: Promela and Spin[1]

Solutions to selected exercises (◊) are provided at the end of this document. Important: You should first try solving them before looking at the solutions. You will otherwise learn **nothing**.

1 Basic Promela

Exercise 1. Do all interleavings of the following program ensure that x is always between 0 and 200? Encode it in Promela and use Spin to determine the answer.

```
x = 0;
```

```
Thread.start { // P    Thread.start { // Q    Thread.start { // R
  while (true) {         while (true) {         while (true) {
    if (x<200)             if (x>0)               if (x==200)
        x++;                   x--;                   x=0;
  }                      }                      }
}                      }                      }
```

Exercise 2. (◊) Implement the following entry/exit protocol (Attempt I) seen in class, in Promela. Use the Promela code in the slides as an aid in understanding Promela syntax.

```
int turn = 1;
```

```
Thread.start { // P          Thread.start { // Q
  while (true) {               while (true) {
     await (turn==1);             await (turn==2);
     turn = 2;                    turn = 1;
  }                             }
}                            }
```

Exercise 3. Implement the following entry/exit protocol (Attempt III) seen in class, in Promela. Use the Promela code in the slides as an aid in understanding Promela syntax.

```
boolean wantP = false;
boolean wantQ = false;
```

```
Thread.start { //P                Thread.start { // Q
  while (true) {                    while (true) {
   // non-critical section           // non-critical section
   wantP = true;                     wantQ = true;
   await (!wantQ);                   await (!wantP);
   // CRITICAL SECTION               // CRITICAL SECTION
   wantP = false;                    wantQ = false;
   // non-critical section           // non-critical section
  }                                 }
}                                 }
```

[1] Sources include: http://www.cs.toronto.edu/~chechik/courses01/csc2108/lectures/spin.2up.pdf

Exercise 4.

Draw the transition system of the following two programs separately and then compare them (note: you can use SpinSpider as a guide):

```
% Program 1
  byte state = 1;
  active proctype A(){
    atomic {
      (state==1) ->
        state = state+1
    }
  }
  active proctype B() {
    atomic {
      (state==1) ->
        state = state-1
    }
  }
```

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

Exercise 5. Check whether the following algorithm guarantees mutual exclusion by adding an auxiliary variable `critical` and appropriate statements. Do you recognize this algorithm?

```
bool flag[2]
bool turn

active [2] proctype user()
{
    flag[_pid] = true
    turn = _pid
    (flag[1-_pid] == false || turn == 1-_pid)

crit:   skip    // critical section

    flag[_pid] = false
}
```

Exercise 6. What happens in the previous algorithm if you exchange the line `flag[_pid]= true` with the line `turn = _pid`?

Exercise 7. Consider the following simplified presentation of the Bakery Algorithm for two processes:

```
int np,nq =0;
Thread.start { // P
  while (true) {
   // non-critical section
   np = nq + 1;
   await nq==0 or np<=nq;
   // CRITICAL SECTION
   np = 0;
   // non-critical section
  }
}
```

```

Thread.start { // Q
  while (true) {
   // non-critical section
   nq = np + 1;
   await np==0 or nq<np;
   // CRITICAL SECTION
   nq = 0;
   // non-critical section
  }
}
```

1. Encode it in Promela.

2. We mentioned in the lectures that if assignment is not atomic, then mutual exclusion is not guaranteed. Verify this in spin. Since assignment is atomic in spin, you first must split line 5 (in P and Q) into two assignment operations.

3. Add the command `np = 1` in thread P just before line 4 and add `nq=1` in thread Q just before line 4. Show, using Spin, that the resulting program does enjoy mutual exclusion.

4. If you verify the code above (for mutex), where assignment is assumed atomic, Spin seems to report an assertion violation. Why? Hint: run the following code:

```
byte n=0;

active proctype P() {
  int i=0;
  for (i: 1..1000) {
    n++;
  }
}
```

Exercise 8. The following extension to 3 processes of Dekker's algorithm is known to deadlock.

```
int turn=0
flags = [false, false, false]

3.times {
    int id = it
    Thread.start {
      int left = (id+2)%3;
      int right = (id+1)%3;
      while (true) {
        flags[id] = true;
        while (flags[left] || flags[right])
          if (turn == left) {
            flags[id] = false;
            await (turn==id);
            flags[id] = true;
          }
        }
        // Critical Section
        turn = right;
        flags[id]=false;
      }
}
```

Encode it in Promela and verify where it deadlocks (there are multiple traces that lead to deadlock, exhibiting one suffices). Here is some code that you can start from.

```
byte turn;
bool flags[3];


proctype P() {
  byte myId = _pid-1;
  /* complete here */
}

init {
  turn=0;
  byte i;
  for(i:0..2) { flags[i] = false; }
  atomic {
    for (i:0..2) { run P(); }
  }
}
```

Exercise 9. (◊)

1. Define a general semaphore with operations `acquire(sem)` and `release(sem)`, where `sem` is a numeric shared global variable. Since there are no functions in Promela, you can declare macros using `inline`. Hint: use `atomic` and blocking expressions.

2. Check that your solution is correct by providing an example of its use.

Exercise 10. Implement the Bar exercise from eb5, in Promela, using the semaphore encodings from Exercise. 9. Use Spin to show that the required invariant is upheld, i.e. that only one Jets fan goes in for every two Patriot fans, by inserting an appropriate assertion. In your example use 20 Jets fans and 20 Patriot fans. Here is a stub:

```
byte ticket = 0;
byte mutex = 1;
/* additional declarations  here */




inline acquire(sem) {
  atomic {
    sem>0 -> sem--
  }
}

inline release(sem) {
  sem++
}

active [20] proctype Jets() {
  /* complete */
}


active [20] proctype Patriots() {
  /* complete */
}
```

Exercise 11. Implement the Car Wash exercise from eb5, in Promela, using the semaphore encodings from Exercise. 9. Use Spin to show that there is at most one car in any of the three stations. Here is a stub:

```
#define N 3 /* Number of Washing Machines */
#define C 10 /* Number of Washing Machines */

byte permToProcess[N]
byte doneProcessing[N]
byte station0 = 1
byte station1 = 1
byte station2 = 1
```

```


inline acquire(s) {
 atomic {
  s>0 -> s--
 }
}

inline release(s) {
 s++
}

proctype Car() {
  /* complete */

}

proctype Machine(int i)  {
  /* complete *


}

init {
  byte i;

  for (i:0..(N-1)) {
     permToProcess[i]=0;
     doneProcessing[i]=0;
   }

 atomic {
   for (i:1..(C)) {
     run Car();
   }
   for (i:0..(N-1)) {
     run Machine(i);
   }
 }
}
```

2 Channels

Exercise 12. (◊) What does the following program print?

```
proctype A(chan q1) {
  chan q2;
  q1?q2;
  q2!123
}
proctype B(chan qforb) {
  int x;
  qforb?x;
  printf("x=%d\n",x)
}
init {
  chan qname = [1] of { chan };
  chan qforb = [1] of { int };
  run A(qname);
  run B(qforb);
  qname!qforb
}
```

Exercise 13. ($\Diamond$) Implement a binary semaphore in Promela using message passing and synchronous communication. You should have two proctypes, `semaphore` and `user`. Here is the code for the user:

```
#define acquire 0
#define release 1
chan sema = [0] of { bit };  /* synchronous channel */
proctype semaphore() {
  /* complete this */
}
proctype user() {
  do
  :: sema?acquire;
     /* crit. sect */
     sema!release;
     /* non-crit. sect. */
  od
}
init {
  run semaphore();
  run user();
  run user();
}
```

Exercise 14. Implement the turnstile example from class using channels. Here is some code to get you started:

```
mtype { bump , read  };
chan counter = [0] of { mtype, chan }
chan reply[2] = [0] of { byte }

proctype Counter() {
   /* complete */
}

proctype Turnstile() {
  /* complete */
}

init {
  byte x;
  atomic {
    run Counter();
    run Turnstile();
    run Turnstile();
  }
  _nr_pr==2;
  counter!read(reply[0]);
  reply[0]?x;
  printf("%d\n",x) /* should print 20 */
}
```

3 Solutions to Selected Exercises

Answer to exercise 2

```
byte turn=1;

active proctype  P() {
  do
     :: do
          :: turn==1 -> break;
          :: else
        od;
        printf("P went in \n");
        turn = 2
  od
}

active proctype Q() {
  do
    :: do
          :: turn==2 -> break;
          :: else
        od;
       printf("Q went in \n");
       turn = 1
       od
}
```

The following variation is acceptable but no quite right since the “await” does busy-waiting, it does not block:

```
byte turn=1;

active proctype  P() {
  do
    :: turn==1;
       printf("P went in \n");
       turn = 2
  od
}

active proctype Q() {
  do
    :: turn==2;
       printf("Q went in \n");
       turn = 1
       od
}
```

Answer to exercise 9

```
inline acquire(sem) {
  atomic {
    sem>0;
    sem--
  }
}
inline release(sem) {
sem++
}
```

Answer to exercise 12

- Init sends qforb to A on qname
- A sends 123 to qforb
- B receives 123 on qforb and prints it

Answer to exercise 13

```
#define acquire 0
#define release 1
chan sema = [0] of { bit };
proctype semaphore() {
  byte count = 1;
  do
  :: (count == 1) -> sema!acquire; count = 0
  :: (count == 0) -> sema?release; count = 1
  od
}

proctype user() {
  do
  :: sema?acquire;
     /* crit. sect */
    sema!release;
     /* non-crit. sect. */
  od
}
init {
  run semaphore();
  run user();
  run user();
}
```