CS 6110 — Assignment 1

Gavin Gray u1040250

1. (30 pts) Run the program in Book 1, Exercises 21.4 (Page 396) on "Bubble sorting," and see how the bug is discovered (the "sortedness assertion" fails). Fully explain all uses of nondeterminism in this Promela model. How does it help attain full state-space coverage of the finite-state model?

   Student Response: The first use of non-determinism is when initializing the values of the array 'a'. This is done by using the repetitive 'do' statement without guards on the separate branches. This will non-deterministically choose a branch to evaluate and will stop when the "break" is hit. In the search space, this will generate all possible inputs for an N-sized array (in our case $N = 4$).

   The second use of non-determinism is when checking the "sortedness" of the array. This time there are guards on the separate branches within the do-statement, however, if multiple evaluate to 'true' then one is chosen non-deterministaclly. This implies that without the index 't' incriminating and decrementing, we could skip over the error altogether! This ensures that we do in fact search he entire array for incorrectly sorted values.

2. (30 pts) Change this Bubblesort to a working version; verify that it works now, removing its present bug (plus others you might notice). Enter it as a Promela program, run it under SPIN, and show that the "sortedness assertion" now passes.

   Student Response: A simple fix for the bubble sort can be made. Instead of testing that $t == a[1]$, we can simply keep a counter of the number of swaps made, and keep swapping until the number of swaps is zero (for reference the swap counter is labeled 'c' in my promela code). The fixed promela code can be found at the end of this document with the title *bubble_sort_fixed.pml*.

3. (40 pts) Change the example in 21.2 as follows:

   (a) (20 pts) Instead of asking the right fork "are you free" and then taking an "ack" or a "nack," try to grab the left fork atomically (via rendezvous) and then the right fork. A piece of plausible Promela is given below; finish its missing parts. Type this in, and verify that it deadlocks. Steps to run SPIN in the shell are below. Interactive runs will be demo'ed in class.

   Student Response: You will find the finished code below in the file *phil_deadlock.pml*. We can observe the deadlock by running this Promela code in SPIN and seeing that the output says

   `'Error: 1 invalid end state'`

   This is achieved when all philosophers have grabbed the left fork and are in the state of trying to grab the right fork (which can't be done). An error trail is given below, the lines indicated describe the deadlock and invalid end state of the program.

```
7: proc  6 (phil:1) phil_deadlock.pml:4 (state 1) [lfp!0]
9: proc  6 (phil:1) phil_deadlock.pml:4 (state 2) [rfp!0]
Eating 11: proc  6 (phil:1) phil_deadlock.pml:4 (state 3)
    [printf('Eating')]
12: proc  6 (phil:1) phil_deadlock.pml:4 (state 4) [lfv!0]
14: proc  4 (phil:1) phil_deadlock.pml:4 (state 1) [lfp!0] <--
16: proc  6 (phil:1) phil_deadlock.pml:4 (state 5) [rfv!0]
18: proc  6 (phil:1) phil_deadlock.pml:4 (state 1) [lfp!0] <--
20: proc  5 (phil:1) phil_deadlock.pml:4 (state 1) [lfp!0] <--
```

   (b) (30 pts – 10 pts each for deadlock, communal progress, indiv progress) A way to avoid deadlock is to have one philosopher grab the right first, then the left (all others grab left first then right). Make this change. Does this prevent deadlocks? Does it allow communal progress (at any point, some philosopher is able to eventually eat). Does it allow individual progress by each philosopher? (For this, capitalize on the problem symmetry and check one member of a symmetric set of processes.) Note: After checking for deadlocks, put suitable never automaton checks for these "progress" assertions and verify. (To keep solutions uniform, remove progress labels and rely on never automata.)

   Student Response: The associated code is found below labeled *phil_resolved.pml* and the philosopher whose forks are reversed (i.e. picks up the right fork first) is labeled. Performing this flip on only one philosopher does indeed prevent deadlock

as the program terminates in a valid state. To check for progress (communal and individual) we can insert a boolean flag, mine is labeled '*progress*', that should continuously go from $0 \longrightarrow 1$ for communal progress. Additionally, to check individual progress, we can say that if this flag is flickering, then due to problem symmetry individual progress is being made by the three philosophers. To claim these two properties, we put the *never automaton* in our code which is as follows:

```
never {
  do
  :: 1
  :: !progress -> goto accept;
  od
  accept: !progress -> goto accept;
}
```

```
/* fixed.pml */
#define Size 5
#define aMinIndx 1
#define aMaxIndx (Size-1)
/* Gonna "waste" a[0] because Sedgewick uses 1-based arrays*/
active proctype bubsort()
{
  byte j, t, c; /* <- NEW */
  bit a[Size];
  /* Nondeterministic array initialization */
  do ::break ::a[1]=1 ::a[2]=1 ::a[3]=1 ::a[4]=1 od;

  t=a[aMinIndx];
  j=aMinIndx+1;

  do
  :: (j > (aMaxIndx)) -> break
  :: (j <= (aMaxIndx)) ->
    if
    :: (a[j-1] > a[j]) -> t=a[j-1]; a[j-1]=a[j]; a[j]=t; c++ /* <- NEW */
    :: (a[j-1]<= a[j])
    fi;
    j++
  od;

  do
  :: c!=0 -> t=a[aMinIndx]; j=aMinIndx+1; c=0 /* <- NEW */
    do
    :: (j > (aMaxIndx)) -> break
    :: (j <= (aMaxIndx)) ->
      if
      :: (a[j-1] > a[j]) -> t=a[j-1]; a[j-1]=a[j]; a[j]=t; c++ /* <- NEW */
      :: (a[j-1] <= a[j])
      fi;
    j++
    od;
  :: c==0 -> break /* <- NEW */
  od;
```

```
  /*  Comb from location-1 to look for sortedness  */
  t=1;
  do
  :: t < aMaxIndx-1 -> t++
  :: t > aMinIndx -> t--
  :: a[t] > a[t+1] -> assert(0)
  od;
}
/* End Of File */
```

```
/* phil_deadlock.pml */
proctype phil(chan lfp, lfv, rfp, rfv)
{ do
  :: lfp!0 -> rfp!0 -> printf("Eating") -> lfv!0 -> rfv!0
  od
}
// This is a mutex with the P and V operations
// https://en.wikipedia.org/wiki/Semaphore_(programming)
//
proctype fork(chan p, v)
{ do
  :: p?0 // Fork taken!
  -> v?0 // Fork returned!
od
}
init {
  chan p0 = [0] of { bit };
  chan v0 = [0] of { bit };
  chan p1 = [0] of { bit };
  chan v1 = [0] of { bit };
  chan p2 = [0] of { bit };
  chan v2 = [0] of { bit };
  atomic {
    run fork(p0, v0);
    run fork(p1, v1);
    run fork(p2, v2);
    // Connect and run Phil P0 to grab
    // fork 0 on left, then
    // fork 2 on right
    run phil(p0, v0, p2, v2);
    // Connect and run Phil P1 to grab
    // fork 1 on left
    // fork 0 on right
    run phil(p1, v1, p0, v0);
    // Connect and run Phil P2 to grab
    // fork 2 on left
    // fork 1 on right
    run phil(p2, v2, p1, v1);
```

```
  }
}
/* End Of File */
```

```
/* phil_resolved.pml */
byte progress;
proctype phil(chan lfp, lfv, rfp, rfv)
{ do
  :: lfp!0 -> rfp!0 ->
    progress = 1 ->
    printf("Eating") ->
    progress = 0 ->
    lfv!0 -> rfv!0
  od
}
proctype fork(chan p, v)
{ do
  :: p?0 // Fork taken!
  -> v?0 // Fork returned!
od
}
init {
  chan p0 = [0] of { bit };
  chan v0 = [0] of { bit };
  chan p1 = [0] of { bit };
  chan v1 = [0] of { bit };
  chan p2 = [0] of { bit };
  chan v2 = [0] of { bit };
  atomic {
    run fork(p0, v0);
    run fork(p1, v1);
    run fork(p2, v2);
    // Connect and run Phil P0 to grab
    // fork 0 on left, then
    // fork 2 on right
    run phil(p0, v0, p2, v2);
    // Connect and run Phil P1 to grab
    // fork 1 on left
    // fork 0 on right
    run phil(p1, v1, p0, v0);
    // Connect and run Phil P2 to grab
    // fork 2 on left
```

```
    // fork 1 on right
    run phil(p1, v1, p2, v2); // Reversed!
  }
}
never {
  do
  :: 1
  :: !progress -> goto accept;
  od
  accept: !progress -> goto accept;
}
/* End Of File */
```