Heriot-Watt University
School of Mathematical and Computer Sciences

Distributed Systems Programming F21DS1

Possible Solutions to SPIN Exercise Sheet 1

Andrew Ireland

Exercise 1

The order in which the words "Hello" and "World" are printed is not fixed. That is, the outcome is nondeterministic so using different seed values will eventually lead to different orderings of the words.

Exercise 2

The model is non-deterministic, *i.e.* the final values of the variables `x` and `y` may be 2 and 5 respectively OR 2 and 4 respectively.

Exercise 3

In the first version the final value of `state` may be 0, 1 or 2. While in the second version the final value of `state` will be either 0 or 2.

Exercise 4

```
mtype = { done };

chan hello_to_control = [0] of { mtype };
chan control_to_world = [0] of { mtype };

proctype hello(chan x){ printf("Hello\n"); x!done; }

proctype world(chan x){ x?done; printf("World\n"); }

proctype control(chan x, y){ x?done; y!done; }

init { atomic { run world(control_to_world);
                run hello(hello_to_control);
                run control(hello_to_control, control_to_world) } }
```

Exercise 5

`division` is a tail-recursive procedure, i.e. on termination (base case) of the recursion the result is available and is communicated directly back to the `init` process. In the case of

`fact`, after each recursive call further computation is required therefore the communication of intermediate results between each recursive call is required.

Exercise 6

```
mtype = { coin20, coin50, milk, plain };

chan coin_channel = [1] of { mtype };
chan choc_channel = [1] of { mtype };

proctype customer(){
         do
         :: coin_channel!coin20;
            choc_channel?milk;
         :: coin_channel!coin50;
            choc_channel?plain;
         od
}

proctype vender(){
         do
         :: coin_channel?coin20;
            choc_channel!milk;
         :: coin_channel?coin50;
            choc_channel!plain;
         od
}

init { atomic { run vender(); run customer(); }}
```

Exercise 7

```
mtype = { milk, plain };

chan coin_channel = [1] of { byte };
chan choc_channel = [1] of { mtype };

byte milk_bars = 10, plain_bars = 5;
int  coin_box = 0;

proctype customer(){
         do
         :: coin_channel!20;
            choc_channel?milk;
         :: coin_channel!50;
            choc_channel?plain;
         od
}

proctype vender(){
```

```
        do
        :: ((milk_bars > 0) && coin_channel?[20]) ->
                              coin_channel?20;
                              choc_channel!milk;
                              milk_bars = milk_bars-1;
                              coin_box = coin_box + 20;
        :: ((plain_bars > 0) && coin_channel?[50]) ->
                               coin_channel?50;
                               choc_channel!plain;
                               plain_bars = plain_bars-1;
                               coin_box = coin_box + 50;
        od
}

init { atomic { run vender(); run customer(); }}
```

Exercise 8

```
mtype = { milk, plain };

chan coin_channel = [1] of { byte };
chan choc_channel = [1] of { mtype };

byte milk_bars = 10, plain_bars = 5;
int  coin_box = 0;

proctype customer(){
        do
        :: coin_channel!20;
           choc_channel?milk;
        :: coin_channel!50;
           choc_channel?plain;
        od
}

proctype vender(){

        assert(coin_box == (450-(milk_bars*20 + plain_bars*50)));
        do
        :: ((milk_bars > 0) && coin_channel?[20]) ->
                              coin_channel?20;
                              choc_channel!milk;
                              milk_bars = milk_bars-1;
                              coin_box = coin_box + 20;
                              assert(coin_box == (450-(milk_bars*20 + plain_bars*50)));
        :: ((plain_bars > 0) && coin_channel?[50]) ->
                               coin_channel?50;
                               choc_channel!plain;
                               plain_bars = plain_bars-1;
```

```
                                coin_box = coin_box + 50;
                                assert(coin_box == (450-(milk_bars*20 + plain_bars*50)));
        od
}

init { atomic { run vender(); run customer(); }}
```

Exercise 9

Note that the system assertion `budget + coin_box == 450` can not be represented via a monitor process. This is because the invariant is only true after each transaction. For this reason the assertion is tested before the `vender` process enters its loop and after every iteration (transaction) is complete.

```
mtype = { milk, plain };

chan coin_channel = [1] of { byte };
chan choc_channel = [1] of { mtype };

int budget = 450, coin_box = 0;

proctype customer(){
        do
        :: coin_channel!20;
           choc_channel?milk;
        :: coin_channel!50;
           choc_channel?plain;
        od
}

proctype vender(){
        byte milk_bars = 10, plain_bars = 5;

        assert(budget+coin_box == 450);
        do
        :: ((milk_bars > 0) && coin_channel?[20]) -> coin_channel?20;
                            coin_box = coin_box + 20;
                            budget=budget-20;
                            choc_channel!milk;
                            milk_bars = milk_bars-1;
                            assert(budget+coin_box == 450);
        :: ((plain_bars > 0) && coin_channel?[50]) -> coin_channel?50;
                            coin_box = coin_box + 50;
                            budget=budget-50;
                            choc_channel!plain;
                            plain_bars = plain_bars-1;
                            assert(budget+coin_box == 450);
        od
}

init { atomic { run vender(); run customer(); }}
```

Exercise 10

The water storage system, as described in the exercise sheet, is not sophisticated enough to ensure that the water level is always within the range 20 to 30 units. Basically there is no guarantee that the sensors will detect low or high water levels before a violation occurs. This is demonstrated by the following model. Try evaluating the model using XSPIN.

```
mtype = { open, close }

mtype out=open, in=close;
byte  water_level = 25;
byte  user_water  = 0;  /* user reservoir */

proctype sensors(){
         do
         :: atomic{ (water_level <= 20) -> out=close; in=open; }
         :: atomic{ (water_level >= 30) -> out=open; in=close; }
         od
}

proctype user(){
         do
         :: (user_water > 0) -> user_water=user_water-1;
         :: true             -> skip;
         od
}

proctype inlet(){
         do ::(in == open) -> water_level=water_level+1; od
}

proctype outlet(){
         do ::(out == open) -> atomic{ water_level=water_level-1;
                                       user_water=user_water+1;}
         od
}

proctype monitor(){ do :: assert( water_level >= 20 &&
                                  water_level <= 30 ); od }

init{ atomic{ run monitor(); run sensors();
              run user(); run inlet(); run outlet(); } }
```