MP 6 – Modeling in Promela and SPIN

CS 477 – Spring 2018
Revision 1.0

Assigned April 23, 2018
Due May 2, 2018, 9:00 PM
Extension 48 hours (penalty 20% of total points possible)

1 Change Log

1.0 Initial Release.

2 Objectives and Background

The purpose of this MP is to test the student's ability to

- build a model for a system in SPIN
- use SPIN and LTL to specify and verify a system based on English specifications

Another purpose of MPs and HWs in general is to provide a framework to study for the exam. Several of the questions on the exam will appear similar to HW and MP problems. A final purpose for MPs is to give you ideas that can help with fourth credit projects.

3 Turn-In Procedure

The pdf for this assignment (`mp6.pdf`) should be found in the `mps/mp6/` subdirectory of your svn directory for this course. You should put code answering the problem below in the file `mp6.pml`. You will also be asked to create three files containing LTL formulae, `ltl1.pml`, `ltl2.pml`, and `ltl3.pml`. Your completed `mp6.pml`, `ltl1.pml`, `ltl2.pml`, and `ltl3.pml` files should be put in the `mps/mp6/` subdirectory of your svn directory (where `mp6.pdf`) was originally found) and committed as follows:

```
svn add mp6.pml ltl1.pml ltl2.pml ltl3.pml
svn commit -m "Turning in mp6"
```

4 Modeling in Promela

Recollect the example of the candy machine given as an example of a labeled transition system in the slides `http://courses.engr.illinois.edu/cs477/sp2018/lectures/19-lts.pdf`.
In that example, we gave a description of the candy machine, but not of the customers who would interact with it. Suppose we wish to model a set of honest customers, each of whom wants to buy some number of just one type of candy, KitKat or MarsBar, respectively. The following is Promela code that can model this in the case where there are three customers, two of whom want KitKats, (two and one), and one customer who wants one MarsBar, for a total of four candies wanted:

```
/* File: candy.pml */

mtype {Pay, KitKat, MarsBar};

//byte kcoins = 2;
//byte mcoins = 2;
bool machine_done = 0;
byte candies_paid = 0;
byte candies_delivered = 0;

chan slot = [2] of {mtype}
chan candy_choice = [1] of {mtype}
chan candy_tray = [4] of {mtype}

proctype Kcustomer(byte id; byte kcoins){
  {do
   :: kcoins != 0;
      atomic{slot ! Pay;
             printf("Payment made by Kcustomer %d\n", id);
             kcoins = kcoins - 1;
             candies_paid = candies_paid + 1;
             printf("candies_paid == %d\n", candies_paid)};
      atomic{candy_choice ! KitKat;
             printf("Kcustomer %d chose a KitKat.\n", id)};
      atomic{candy_tray ? KitKat;
             printf("Kcustomer %d took a KitKat\n", id)}
   :: kcoins == 0; goto out
   od}
out:  printf("Kcustomer %d is done\n", id)
}

proctype Mcustomer(byte id; byte mcoins){
  {do
   :: mcoins != 0;
      atomic{slot ! Pay;
             printf("Payment made by Mcustomer %d\n", id);
             mcoins = mcoins - 1;
             candies_paid = candies_paid + 1;
             printf("candies_paid == %d\n", candies_paid)};
      atomic{candy_choice ! MarsBar;
             printf("Mcustomer %d chose a MarsBar.\n", id)}
      atomic{candy_tray ? MarsBar;
             printf("Mcustomer %d took a MarsBar\n", id)}
   :: mcoins == 0; goto out
   od}
out: printf("Mcustomer %d is done\n", id)
}

proctype machine () {
  mtype choice;
  {do
```

```
   :: slot ? Pay;
      printf("Received payment\n");
      candy_choice ? choice;
      atomic{if
             ::{choice == KitKat;
                printf("Delivering a KitKat\n")}
             ::{choice == MarsBar;
                printf("Delivering a MarsBar\n")}
             fi;
             candies_delivered = candies_delivered + 1};
      candy_tray ! choice;
      atomic{if
             :: {choice == KitKat;
                 printf("Please take KitKat from tray.\n")}
             :: {choice == MarsBar;
                 printf("Please take MarsBar from tray.\n")}
             fi}
   :: timeout -> goto out
   od}
out:  machine_done = 1; printf("Candy machine is done\n")
}

active proctype monitor () {
  assert ((candies_paid - candies_delivered) >= 0)
  // never give out more than has already been paid for
}

init {
run machine();
run Kcustomer(0,2);
run Kcustomer(1,1);
run Mcustomer(0,1)
}
```

If we run SPIN in simulator mode, we can see a sample behavior as follows:

```
bash-3.2$ spin candy.pml
                      Payment made by Kcustomer 1
                      candies_paid == 1
                      Kcustomer 1 chose a KitKat.
                  Payment made by Kcustomer 0
                  candies_paid == 2
              Received payment
                  Kcustomer 0 chose a KitKat.
                          Payment made by Mcustomer 0
                          candies_paid == 3
              Delivering a KitKat
                  Kcustomer 0 took a KitKat
              Please take KitKat from tray.
                  Payment made by Kcustomer 0
                  candies_paid == 4
              Received payment
              Delivering a KitKat
```

```
                        Mcustomer 0 chose a MarsBar.
                Kcustomer 1 took a KitKat
        Please take KitKat from tray.
        Received payment
                Kcustomer 1 is done
            Kcustomer 0 chose a KitKat.
        Delivering a MarsBar
        Please take MarsBar from tray.
                        Mcustomer 0 took a MarsBar
        Received payment
        Delivering a KitKat
                        Mcustomer 0 is done
            Kcustomer 0 took a KitKat
        Please take KitKat from tray.
            Kcustomer 0 is done
  timeout
        Candy machine is done
6 processes created
bash-3.2$
```

This still leaves us needing to know whether our code's behavior is always correct: Does the machine always get paid before it gives out candy, and do the customers get all the candy they want? We can check that the machine always gets paid first by checking that the assert inside the monitor process is always true. The second condition, that the customers get all the candy they want can be checked by checking that every process terminates correctly. We can do both of these by using spin in its model checking mode:

```
bash-3.2$ spin -a candy.pml
bash-3.2$ gcc -o pan pan.c
bash-3.2$ ./pan

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

Full statespace search for:
never claim          - (none specified)
assertion violations +
acceptance   cycles  - (not selected)
invalid end states +

State-vector 88 byte, depth reached 60, errors: 0
     3129 states, stored
     2141 states, matched
     5270 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

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

```
unreached in proctype Kcustomer
(0 of 21 states)
unreached in proctype Mcustomer
(0 of 21 states)
unreached in proctype machine
(0 of 32 states)
unreached in proctype monitor
(0 of 2 states)
unreached in init
(0 of 5 states)

pan: elapsed time 0 seconds
bash-3.2$
```

The absence of complaint about assertion violations or invalid end states tells us that we have verified the properties we set out.

We can also use an ltl formula to describe safety. From the customers perspective, the candy machine is "safe" if the customers always receive the candy for they have paid. Notice that in the trace given above, Kcustomer 1 was the first to pay for and select a KitKat, but Kcustomer 0 was the first to take a KitKat. Still, everything ended up alright in the end because both Kcustomers got all the candy they paid for by hte time the machine stop dispensing. To rephrase this as the absence of bad, we can say that it's never the case that the eventually the machine stops delivering candy but the customers have paid for more candy than has been delivered. Thus the hazard to be avoided for the customers may be stated as:

```
<> (machine_done && candies_paid > candies_delivered)
```

which we have put in the file `candyneverltl.pml`. We may convert this into a "never" claim and store the result in the file `candynever.pml` by the command:

```
spin -F candyneverltl.pml > candynever.pml
```

The resultant "never" claim is the following:

```
never  {    /* <> (machine_done && candies_paid > candies_delivered)  */
T0_init:
        do
        :: atomic { ((machine_done && candies_paid > candies_delivered)) -> assert(!((machine_done &&
        :: (1) -> goto T0_init
        od;
accept_all:
        skip
}
```

If we then build and execute the verfier by

```
spin -a -N candynever.pml candy.pml
gcc -o pan pan.c
./pan -a
```

we get the output

```
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)

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

```
Full statespace search for:
never claim         + (never_0)
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 96 byte, depth reached 121, errors: 0
     3129 states, stored
     2142 states, matched
     5271 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

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


unreached in proctype Kcustomer
(0 of 21 states)
unreached in proctype Mcustomer
(0 of 21 states)
unreached in proctype machine
(0 of 32 states)
unreached in proctype monitor
(0 of 2 states)
unreached in init
(0 of 5 states)
unreached in claim never_0
./candynever.pml:9, state 10, "-end-"
(1 of 10 states)

pan: elapsed time 0.01 seconds
bash-3.2$
```

We see that this state (`never_0` state 10, `-end-`) in fact is unreachable. Looking at the code in `candynever.pml`, we see that means we never finish the do-loop. Looking at the do-loop, we see one choice that can always be executed and sends us back to the start of the never claim. The other choice atomically tests if the condition `machine_done && candies_paid > candies_delivered` is true, and it is then it asserts that it is not. If this choice were ever taken, which it would have to be for the LTL formula in `candyneverltl.pml` to be satisfied, the assert would have triggered a failure, and `pan` would have informed us of an assertion violation. Since no assertion violations were reported, the LTL formula is not satisfied by any execution of our system.

Let us now consider when an error is reported. Suppose we believe that `candies_paid` should never have a value of 2. Here the bad thing then is `candies_paid` being equal to 2, except that we don't want it to be equal to 2 at any point in any execution, not just not at the start of any execution. We can capture this negative property with the LTL formula

```
<> (candies_paid == 2)
```

If we store this is a file `testltl.pml`, and save its conversion to a "never" claim in `testnever.pml` and compile and execute the verifier for this case via

```
spin -F testltl.pml > testnever.pml
spin -a -N testnever.pml candy.pml
gcc -o pan pan.c
./pan -a
```

we get the output

```
bash-3.2$ ./pan -a
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
pan:1: assertion violated  !((candies_paid==2)) (at depth 20)
pan: wrote candy.pml.trail

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

Full statespace search for:
never claim          + (never_0)
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 96 byte, depth reached 20, errors: 1
       11 states, stored
        0 states, matched
       11 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.001 equivalent memory usage for states (stored*(State-vector + overhead))
    0.281 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 seconds
bash-3.2$
```

This time you may note we are told we have an assertion failure and that there is a trail file. We may examine that trail file as follows:

```
bash-3.2$ spin -t -p candy.pml
starting claim 5
spin: couldn't find claim 5 (ignored)
using statement merging
Starting machine with pid 3
  2: proc  1 (:init::1) candy.pml:80 (state 1) [(run machine())]
```

```
Starting Kcustomer with pid 4
  4: proc  1 (:init::1) candy.pml:81 (state 2) [(run Kcustomer(0,2))]
  6: proc  3 (Kcustomer:1) candy.pml:17 (state 1) [((kcoins!=0))]
Starting Kcustomer with pid 5
  8: proc  1 (:init::1) candy.pml:82 (state 3) [(run Kcustomer(1,1))]
 10: proc  4 (Kcustomer:1) candy.pml:17 (state 1) [((kcoins!=0))]
Starting Mcustomer with pid 6
 12: proc  1 (:init::1) candy.pml:83 (state 4) [(run Mcustomer(0,1))]
 14: proc  5 (Mcustomer:1) candy.pml:34 (state 1) [((mcoins!=0))]
 16: proc  5 (Mcustomer:1) candy.pml:35 (state 2) [slot!Pay]
                              Payment made by Mcustomer 0
 16: proc  5 (Mcustomer:1) candy.pml:36 (state 3) [printf('Payment made by Mcustomer %d\\n'
 16: proc  5 (Mcustomer:1) candy.pml:37 (state 4) [mcoins = (mcoins-1)]
 16: proc  5 (Mcustomer:1) candy.pml:38 (state 5) [candies_paid = (candies_paid+1)]
                              candies_paid == 1
 16: proc  5 (Mcustomer:1) candy.pml:39 (state 6) [printf('candies_paid == %d\\n',candies_p
 18: proc  5 (Mcustomer:1) candy.pml:40 (state 8) [candy_choice!MarsBar]
                              Mcustomer 0 chose a MarsBar.
 18: proc  5 (Mcustomer:1) candy.pml:41 (state 9) [printf('Mcustomer %d chose a MarsBar.\\n
 20: proc  4 (Kcustomer:1) candy.pml:18 (state 2) [slot!Pay]
                             Payment made by Kcustomer 1
 20: proc  4 (Kcustomer:1) candy.pml:19 (state 3) [printf('Payment made by Kcustomer %d\\n'
 20: proc  4 (Kcustomer:1) candy.pml:20 (state 4) [kcoins = (kcoins-1)]
 20: proc  4 (Kcustomer:1) candy.pml:21 (state 5) [candies_paid = (candies_paid+1)]
                             candies_paid == 2
 20: proc  4 (Kcustomer:1) candy.pml:22 (state 6) [printf('candies_paid == %d\\n',candies_p
spin: trail ends after 21 steps
#processes: 6
machine_done = 0
candies_paid = 2
candies_delivered = 0
queue 1 (slot): [Pay][Pay]
queue 2 (candy_choice): [MarsBar]
 21: proc  5 (Mcustomer:1) candy.pml:42 (state 13)
 21: proc  4 (Kcustomer:1) candy.pml:23 (state 10)
 21: proc  3 (Kcustomer:1) candy.pml:18 (state 7)
 21: proc  2 (machine:1) candy.pml:51 (state 29)
 21: proc  1 (:init::1) candy.pml:84 (state 5) <valid end state>
 21: proc  0 (monitor:1) candy.pml:75 (state 1)
6 processes created
bash-3.2$
```

Here we see a trace where the MCustomer 0 pays for a candy, and then the Kcustomer 1 pays for a candy. There is nothing fundamentaly wrong with this behavior, so we must have a misunderstanding somewhere. Maybe we meant for the model to handle `candies_paid` by decrementing it each time a customer picked up their candy. Or maybe we really meant the temporal property

```
<> (candies_paid - candies_delivered == 2)
```

To resolve this, you have to make a descission what each of the variables represents, and then correct either the model or the property as is appropriate.

Now its your turn.

5 Problem

1. (50 pts) This is a modeling and design verification problem.

   Consider how a simple lock on a waterway works. There are two gates, an upstream gate and downstream gate, and there are two valves (typically panels in the gates), an inlet valve on the upstream side and an outlet valve on the downstream side. When the upstream gate and the inlet valve are both closed, no water may flow from upstream into the lock, and when the downstream gate and outlet valve are both closed no water may flow out of the lock. If either the upstream gate or the upstream valve is open and the water level upstream is higher than the water level in the lock, then the water flows into the lock, and if the downstream gate and valve are both shut, the water will rise to the level of the upstream water. Dually, if either the downstream gate or the downstream valve is open and the water level in the lock is higher than the water level downstream, then water flows out of the lock, and if both the upstream gate and valve are closed the water will fall to the level of the downstream water. You may assume (an abstraction) that the water level upstream and downstream are constant with upstream always strictly higher than downstream.

   The initial position of the lock is with the downstream gate open, and the upstream gate and both valves closed, and the water level equal to the downstream level. If a boat approaches from the upstream side (headed downstream), if the upstream gate is open, then the boat goes into the lock, while if the upstream gate is closed, it waits for it to open. If a boat is waiting on the upstream side (headed downstream) and the upstream gate is closed and the downstream gate is open, then the downstream gate closes, and when it is closed, the inlet valve opens. If the inlet valve is open and the water level in the lock is the same as the upstream water level, the inlet valve closes, and when it is closed, the upstream gate opens. When the upstream gate is open, if a boat is waiting on the upstream side (headed downstream), it enters the lock. Once a boat headed downstream has entered the lock, the gate closes and when it is closed, the outlet valve opens. If the outlet valve is open and the water level is equal to the downstream level, then the outlet valve closes, the downstream gate opens, and then the boat exists the lock to the downstream side. A boat that is downstream of the lock headed downstream may either continue to head downstream or turn and head upstream approaching the lock from the downstream side.

   When a boat approaches from the downstream side (headed upstream), if the downstream gate is open, the boat goes into the lock and if the downstream gate is closed, it waits for it to open. If a boat is waiting on the downstream side (headed upstream) and the downstream gate is closed and the upstream gate is open, then the upstream gate closes, and when it is closed the outlet valve opens. If the outlet valve is open and the water level in the lock is the same as the downstream water level, the outlet valve closes, and when it is closed, the downstream gate opens. When the downstream gate is open, if a boat is waiting on the downstream side (headed upstream), it enters the lock. Once a boat headed upstream has entered the lock, the gates close and when they are closed, the inlet valve opens. If the inlet valve is open and the water level is equal to the upstream level, then the inlet valve closes and the upstream gate opens. If there is a boat headed upstream in the lock and the upstream gate is open, then the boat exists the lock to the upstream side. A boat that is upstream of the lock headed upstream may either continue to head upstream or turn and head downstream approaching the lock.

   Write a collection of Promela protocols to model the boat, the water level in the lock, the lock gates and valves in the above scenario. You may assume just one boat exists, and that water levels are measured in integers, and increase or decrease one unit per step of execution. You will need to introduce a collection of variables, but their names should clearly indicate their meaning. For data, you may use the booleans true and false, and the integers, and any enumeration type you clearly define. You should have your model print out each event that occurs (such as the gate opening or a boat entering the lock). Use `atomic` to bind print statements to the statements that do the event. Your model should satisfy that LTL properties given in the next problem. Save your code in the file `mp6.pml`

2. (25 pts) Using the variables and types you introduced in the previous problem, give LTL formulae suitable for producing "never" claims checking the following conditions hold:

   1. (5pts) We never have both gates open at the same time. Put the formula in the file `ltl1.pml`.

2. (5pts) Neither valve is ever open when either gate is open. Put the formula in the file `ltl2.pml`.
3. (8pts) The boat only enters the lock when the water level in the lock is the same as the water level on the side where the boat is. Put the formula in the file `ltl3.pml`.

Your points for these problems will be determined both by whether the formula expresses the given property and by whether your model satisfies the resulting "never" claim.