MP 2 – Modeling

CS 477 – Spring 2013
Revision 1.1

Assigned April 30, 2014
Due May 7, 2014, 9:00 PM
Extension 48 hours (penalty 20% of total points possible)

1 Change Log

1.1 Corrected the Extra Credit problem by adding "with both the gates closed" to the end of the sentence.

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 (`mp2.pdf`) should be found in the `mps/mp2/` subdirectory of your svn directory for this course. You should put code answering the problem below in the file `mp2.pml`. You will also be asked to create three files containing LTL formulae, `ltl1.pml`, `ltl2.pml`, `ltl3.pml`, and `ltl4.pml`. Your completed `mp2.pml`, `ltl1.pml`, `ltl2.pml`, `ltl3.pml`, and `ltl4.pml` files should be put in the `mps/mp2/` subdirectory of your svn directory (where `mp2.pdf`) was originally found) and committed as follows:

```
svn add mp2.pml ltl1.pml ltl2.pml ltl3.pml ltl4.pml
svn commit -m "Turning in mp2"
```

If you do the extra credit problem, you will also need the file `ltlX.pml` and you will need the add it to the files in the `svn add`.

Please read the *Instructions for Submitting Assignments* in

`http://courses.engr.illinois.edu/cs477/mps/index.html`

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/sp2013/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 two 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 each customer wants two candies, for a total of four:

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

mtype {Pay, KitKat, MarsBar}

byte candies_paid = 0;
byte candies_delivered = 0;

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

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

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

proctype machine (byte c) {
  mtype choice;
  {do
   :: printf("Insert coin.\n");
      slot ? Pay;
```

```
      printf("Make choice: KitKat or MarsBar\n");
      candy_choice ? choice;
      atomic{candy_tray ! choice;
             candies_delivered = candies_delivered +1;
             printf("Take candy\n")};
      c = c - 1
   od}
  unless
  {c == 0}
}

active proctype monitor () {
  assert (candies_delivered - candies_paid != 1)
  /* never give out more than has already been paid for */
}

init {
run machine(4);
run Kcustomer(2);
```

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

```
bash-3.2$ spin candy.pml
              Insert coin.
                  Payment made by Kcustomer
                      Payment made by Mcustomer
                  Kcustomer chose a KitKat
              Make choice: KitKat or MarsBar
                      Mcustomer chose a MarsBar
              Take candy
                  Kcustomer took a KitKat
                  Payment made by Kcustomer
              Insert coin.
              Make choice: KitKat or MarsBar
              Take candy
              Insert coin.
                      Mcustomer took a MarsBar
                      Payment made by Mcustomer
              Make choice: KitKat or MarsBar
                      Mcustomer chose a MarsBar
              Take candy
                  Kcustomer chose a KitKat
              Insert coin.
              Make choice: KitKat or MarsBar
              Take candy
5 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
hint: this search is more efficient if pan.c is compiled -DSAFETY

(Spin Version 6.2.4 -- 8 March 2013)
+ Partial Order Reduction

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

State-vector 76 byte, depth reached 70, errors: 0
      703 states, stored
      489 states, matched
     1192 transitions (= stored+matched)
      752 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.070 equivalent memory usage for states (stored*(State-vector + overhead))
    0.284 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.611 memory used for DFS stack (-m10000)
  128.806 total actual memory usage


unreached in proctype Kcustomer
(0 of 18 states)
unreached in proctype Mcustomer
(0 of 18 states)
unreached in proctype machine
(0 of 15 states)
unreached in proctype monitor
(0 of 2 states)
unreached in init
(0 of 4 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 ltl formula to describe safety. From the customers perspective, the candy machine is “safety” if it always gives candy after the customer has paid. (We will omit the kind here, since the program I gave only counts candies delivered in total, and not by kind.) To rephrase this as the absence of bad, we can say that it’s never the case that the customer pays for more candy than is eventually delivered. Thus the hazard to be avoided for the customer may be stated as

```
<> [] (candies_paid - candies_delivered > 0)
```

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  {    /* <> [] (candies_paid - candies_delivered < 0)  */
T0_init:
        do
        :: ((candies_paid - candies_delivered < 0)) -> goto accept_S4
        :: (1) -> goto T0_init
        od;
accept_S4:
        do
        :: ((candies_paid - candies_delivered < 0)) -> goto accept_S4
        od;
}
```

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

```
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)

(Spin Version 6.2.4 -- 8 March 2013)
+ 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 84 byte, depth reached 125, errors: 0
      703 states, stored
      490 states, matched
     1193 transitions (= stored+matched)
      968 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.075 equivalent memory usage for states (stored*(State-vector + overhead))
    0.282 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.611 memory used for DFS stack (-m10000)
  128.806 total actual memory usage
```

```
unreached in proctype Kcustomer
(0 of 19 states)
unreached in proctype Mcustomer
(0 of 19 states)
unreached in proctype machine
(0 of 16 states)
unreached in proctype monitor
(0 of 2 states)
unreached in init
(0 of 4 states)
unreached in claim never_0
./candynever.pml:9, state 10, "(((candies_paid-candies_delivered)<0))"
./candynever.pml:11, state 13, "-end-"
(2 of 13 states)

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

We used the `-a` flag for `./pan` because it contains acceptence states (`accept_S4`) we want to know if it is reached. It is reached precisely when `((candies_paid - candies_delivered < 0))`. We see that this state (`never_0` state 10) in fact is unreachable, which is what we want.

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 21)
pan: wrote candy.pml.trail

(Spin Version 6.2.4 -- 8 March 2013)
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 84 byte, depth reached 21, errors: 1
        8 states, stored
        0 states, matched
        8 transitions (= stored+matched)
        7 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.001 equivalent memory usage for states (stored*(State-vector + overhead))
    0.282 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.611 memory used for DFS stack (-m10000)
  128.806 total actual memory usage



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

Thos 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:) candy.pml:66 (state 1) [(run machine(4))]
                  Insert coin.
  4: proc  2 (machine) candy.pml:47 (state 1) [printf('Insert coin.\\n')]
Starting Kcustomer with pid 4
  6: proc  1 (:init:) candy.pml:67 (state 2) [(run Kcustomer(2))]
Starting Mcustomer with pid 5
  8: proc  1 (:init:) candy.pml:68 (state 3) [(run Mcustomer(2))]
 10: proc  4 (Mcustomer) candy.pml:30 (state 1) [slot!Pay]
                          Payment made by Mcustomer
 11: proc  4 (Mcustomer) candy.pml:31 (state 2) [printf('Payment made by Mcustomer\\n')]
 12: proc  4 (Mcustomer) candy.pml:32 (state 3) [candies_paid = (candies_paid+1)]
                          candies_paid == 1
 13: proc  4 (Mcustomer) candy.pml:33 (state 4) [printf('candies_paid == %d\\n',candies_pa
 15: proc  4 (Mcustomer) candy.pml:34 (state 6) [candy_choice!MarsBar]
                          Mcustomer chose a MarsBar
 16: proc  4 (Mcustomer) candy.pml:35 (state 7) [printf('Mcustomer chose a MarsBar\\n')]
 18: proc  3 (Kcustomer) candy.pml:14 (state 1) [slot!Pay]
                        Payment made by Kcustomer
 19: proc  3 (Kcustomer) candy.pml:15 (state 2) [printf('Payment made by Kcustomer\\n')]
 20: proc  3 (Kcustomer) candy.pml:16 (state 3) [candies_paid = (candies_paid+1)]
                        candies_paid == 2
 21: proc  3 (Kcustomer) candy.pml:17 (state 4) [printf('candies_paid == %d\\n',candies_pa
spin: trail ends after 22 steps
#processes: 5
candies_paid = 2
candies_delivered = 0
```

```
queue 1 (slot): [Pay][Pay]
queue 2 (candy_choice): [MarsBar]
 22: proc  4 (Mcustomer) candy.pml:36 (state 12)
 22: proc  3 (Kcustomer) candy.pml:18 (state 8)
 22: proc  2 (machine) candy.pml:48 (state 2)
 22: proc  1 (:init:) candy.pml:69 (state 4) <valid end state>
 22: proc  0 (monitor) candy.pml:61 (state 1)
5 processes created
bash-3.2$
```

Here we see a trace where the MarsBar customer buys a candy, and then the KitKat customer buys 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. (50pts) Here you are asked to build a Promela model forthe boat problem from the last assignment, except this time you are required to have two boats. The description, adjusted to two boats is as follows:

   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, increasing the level in the lock (one unit per time interval (an abstraction)), and 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, decreasing the level in the lock (one unit per time interval (an abstraction)). 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 (eventually) 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, and if a boat headed upstream is in the lock, it exists. (The lock may hold multiple boats at once.) 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, then 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 it is 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 Promela model for the above scenario, with two boats in addition to the lock and the water. 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 `mp2.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`.
    4. (7pts) A boat headed in a given direction will get to that side of the lock. Put the formula in the file `ltl4.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.

3. (Extra Credit) Using the variables and types you introduced in the previous problem, give an LTL formula suitable for producing a "never" claim checking the following condition hold:

    1. (4pts) There are never two boats in the lock at the same time headed in opposite directions with both the gates closed. Put the formula in the file `ltlX.pml`.