MP 3 – Modeling

CS 477 – Spring 2013

Revision 1.0

Assigned April 12, 2013
Due April 19, 2013, 11:59 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 your system based on English specifications

Another purpose of MPs in general is to provide a framework to study for the exam. Several of the questions on the exam will appear similar to the 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 (`mp3.pdf`) should be found in the `mps/mp3/` subdirectory of your svn directory for this course, along with a skeleton version of the file `mp3.thy`. You should put code answering the problem below in the file `mp3.pml`. Your completed `mp3.pml` file should be put in the `mps/mp3/` subdirectory of your svn directory (where `mp3.pdf`) was originally found) and committed as follows:

```
svn add mp3.pml
svn commit -m "Turning in mp3"
```

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 to 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};
      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};
      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;
             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);
run Mcustomer(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$ cc -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. Now its your turn.

5 Problem

1. (50pts) Consider the following situation:

   My husband, Carl, and I have two sons, who drink a lot of milk, particularly for dinner. When each of us gets home, if we find that the milk is out, we go to the store to buy more. Our refrigerator is small and cannot hold all the milk if we each bought milk. To avoid the unacceptable situation of having more milk than we can fit in the frig because we separately went to get milk, we leave a note on the frig saying “Gone to get milk”, which we take off when we return. Your job is to model this situation in Promela using processes to represent myself, my husband, and my sons (this can be just one). Your solution should match the general behavior described above, and should guarantee two properties: there will always eventually be milk in the frig, and there will never be more milk than can fit in the frig. You may use `atomic` to model reasonable assumptions about human behavior, but the portion of the processes for my husband and myself go to the store to get the milk may not be contained within an `atomic`. You may assume the getting milk from the store always succeeds. Your code should contain sufficient assertions to enable SPIN to automatically verify these conditions, when combined with invalid end state checking.