Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.820 Foundations of Program Analysis

Problem Set 6

Technically, this homework is due on December 9, but we will accept submissions without penalty until December 11 at 5:00PM.

In this problem set, you will apply the model checking principles in both theory and practice.

Problem 1 — CTL* (30 points)

For each of the specifications below, write:

- a CTL formula if one exists or ”NONE” if it cannot be expressed in CTL and
- an LTL formula or ”NONE” if it cannot be expressed in LTL.

For each spec below, we write in parentheses the name of the atomic proposition that corresponds to the action.

Part a: (10 points) If I subscribe to Netflix (subscribe), sooner or later I will either unsubscribe (un) or die (rip). Before that happens, I should be allowed to keep watching (watch) movies, but I can’t watch movies from the grave or after I unsubscribe.

Part b: (10 points) If I rent a movie (rent), the movie has to be returned (return) before I unsubscribe. I can only have one rented movie at a time, and if I die before I return the movie, the movie has to be returned the day after I die.

Part c: (10 points) After I subscribe to Netflix (subscribe), netflix will start charging me a fee (charge); the charge will always happen after the beginning of the month (begin-month) but before the fifth (fifth) and I need to pay it (pay) before the fifteenth (fifteenth). If I don’t do that, I can’t rent movies (rent) ever again.

Problem 2 — Spin(70 points)

For this exercise, you will get to work with the spin model checker. The modelchecker can be downloaded from here:

http://spinroot.com/spin/Man/README.html

There is a tutorial available here.

http://spinroot.com/spin/Doc/SpinTutorial.pdf The tutorial is a little long, but after reading the quickstart guide we have below, the main things you will need to read are pages 12 and 16-18.

Also, the following page contains information about using LTL formulas in SPIN, which will be important for this assignment.

http://spinroot.com/spin/Man/ltl.html

Spin quickstart guide. The input language for SPIN is called Promela. A typical program in Promela creates a fixed number of processes which then run concurrently and interact either via shared variables or via channels. For this homework we will only be using shared variables, so you won't have to worry about channels. As an example of a very simple promela program, consider the code below:

```
int x;

ltl p1 { [] (x >= 0) }

proctype pp1(){
  x = x + 1;
}

proctype pp2(){
  x = x - 1;
}

init{
  x = 0;
  run pp1();
  run pp2();
}
```

The init method in the program creates two processes, pp1 and pp2, which increment and decrement x respectively. The variable x is a global variable defined at the top of the program. For illustration purposes, we have added an ltl property to this file; the operator `[]` is spin's ascii version of a square (the G quantifier)—as a word of caution, spin can be very picky about parenthesis in formulas, so always err on the side of too many parenthesis. Spin will try to prove or disprove that the ltl property holds for all possible interleavings of the two processes. To get started, write the code above into a file called `mytest.spin`.

Running Spin. There are two ways to run spin: simulator or verifier generator. If you run just

```
> spin mytest.spin
```

Spin will run in simulator mode, which means it will try to simulate different executions of the model to try to find assertion violations. However, simulator mode will not take into account your LTL properties; it will only identify assertion violations. If you actually want to verify your ltl property, you need to run in verifier generator mode. The first step is to run

```
> spin -a mytest.spin
```

Spin will generate a custom verifier into a file named pan.c; in order to actually verify the code, you need to compile pan.c and run it.

```
> gcc pan.c -o check
> ./check -a
```

In the case of the example above, the property does not hold for all possible executions, so the verifier will output

```
pan:1: assertion violated  !( !((x>=0))) (at depth 8)
pan: wrote mytest.spin.trail
```

followed by some statistics about the verification process. The file mytest.spin.trail contains all the information necessary to reconstruct the execution that led to the property violation, but it's not particularly readable. If you actually want to see a counterexample trace, you can get one by running the following command:

```
> ./check  -r mytest.spin.trail -v
```

The -r flag tells the verifier to execute the program as directed by the given trail, and the -v flag tells it to show verbose output. If you run this, the verifier will show you step by step how the different threads interacted to produce the property violation.

Note: One flag you may find useful to pass to your `./check` executable is -m. If your model is big, you may get an error like `error:  max search depth too small`, which you can fix by giving a bigger search depth with the -m flag.

Part a: (40 points) Create a Spin model that encodes the program below. Assume that `in`, `buf`, `idx` and `N` are global variables and that N is fixed to 5. Variable `buf` is a buffer of size N. The function `input()` is a non-deterministic function that can either return `noop`, `req1`, or `req2`. You will need to introduce an additional lock variable to model the lock. You can assume idx is initialized to zero (the default in promela).

```
thread1(){
  while(true) {
    in = input();
    if(in != noop){
      while(idx >= N) /*wait */;
      lock();
      buf[idx] = in;
      idx = idx + 1;
      unlock();
    }
  }
}

thead2(){
  while(true) {
    while(idx==0) /*wait*/;
    lock();
    out = buf[idx-1];
    idx = idx - 1;
    unlock();
  }
}
```

Part b: (20 points) Use the spin model checker to answer the following questions:

1. Is it true that when thread 1 reads a `req1` from `input()`, thread 2 eventually sets out equal to `req1`?
2. If thread 1 gets a `req1`, and then eventually it stops getting requests (it gets only `noops`), is it true that thread 2 will eventually get a `req1`?
3. It is possible to have an out-of-bounds array access by either thread?
4. If thread 2 executes infinitely often and thread 1 gets a `req1`, and then eventually it stops getting requests (it gets only `noops`), is it true that thread 2 will eventually get a `req1`?

Deliverable You should turn in a file named `hw6.spin` that contains your model and ltl properties to check for each of the questions above. Your properties should be named `prop1`, `prop2`, `prop3` and `prop4`. If you feel that any of the properties is better expressed with assertions, then you can ommit that property and simply mark the assertions that correspond to that property with a comment that says `propn` where `n` is the property number.