CS 476 Homework #14 Due 10:45am on 12/12

Note: Answers to the exercises listed below should be handed to the instructor *in hardcopy* and in *typewritten form* (latex formatting preferred) by the deadline mentioned above. You should also email to `hildenb2@illinois.edu` the Maude code for both exercises.

1. In this exercise, you are asked to find a *loop invariant* for a particular `IMP` program. Because we know computing loop invariants is tricky (especially if this is your first time) we include an example to show you how it can be done:

```
mod MULTIPLICATION is pr IMP-SEMANTICS .
  op mult      : -> Stmt [ctor] .
  op mult-init : Nat Nat -> Memory .
  var N M : Nat .
  eq mult           = while 0 < n do s := s + m ; n := n - 1 od .
  eq mult-init(N,M) = {[m,M] [n,N] [s,0]} .
endm
```

Note that `N` and `M` are the values assigned to variables `n` and `m`, respectively, in the initial memory. Let `S',N',M'` be the values assigned to the variables `s,n,m`, respectively at some other point in the execution. From the definition of the loop, we see (whenever we are not in-between the variable assignments in the loop body) that it is always the case that `S' + (N' * M')` = `N * M`. This relationship between `S',N',M'` expressed as an equation in this case, and more generally as a formula, is what is called a *loop invariant* for this given loop. To define this invariant, we need to define the following two pieces:

   (a) a state proposition that holds when we are not in-between the assignments, i.e., when we start executing the loop (for the first time or after several iterations), or when we have finished with `skip`.

   (b) a state proposition that holds exactly when `S' + (N' * M')` = `N * M`.

Then (a) can be characterized by state proposition `pgm-or-skip` below and (b) is formalized by state proposition `mult-inv` below.

```
mod EXAMPLE is pr MULTIPLICATION + FACTORIAL . inc MODEL-CHECKER *
     (sort Nat to Nat*,sort NzNat to NzNat*,
      op _+_ to _.+_, op _*_ to _.*_, op _<_ to _.<_, op _<=_ to _.<=_) .
  subsort ImpState < State .
  --- predicates
  op mult-inv    : Nat  -> Prop .
  op pgm-or-skip : Stmt -> Prop .
  --- var decls
  var ST ST' : Stmt . var E : Memory . var S N M T : Nat .
  --- main proposition (a)
  eq (ST   | {[s,S] [n,N] [m,M]}) |= mult-inv(T) = S + (M *Nat N) == T .
  --- helper proposition (b)
  eq (ST   | E) |= pgm-or-skip(ST ) = true  .
  eq (skip | E) |= pgm-or-skip(ST ) = true  .
  eq (ST   | E) |= pgm-or-skip(ST') = false [owise] .
endm
```

Finally, we can model check our formula. In each instance below, if we have initial memory `mult-init(N,M)`, then our invariant looks like:

```
[] (pgm-or-skip(mult) -> mult-inv(N * M))
```

There are two key points then:

(a) in order to prove that the multiplication program is correct, we need the mathematical specification of multiplication!

(b) the invariant for the multiplication program is dependent on the parameters in the initial state of the multiplication function.

Here are some concrete examples below:

```
red modelCheck(mult | mult-init(0,1), [] (pgm-or-skip(mult) -> mult-inv(0))) .
red modelCheck(mult | mult-init(1,0), [] (pgm-or-skip(mult) -> mult-inv(0))) .
red modelCheck(mult | mult-init(7,0), [] (pgm-or-skip(mult) -> mult-inv(0))) .
red modelCheck(mult | mult-init(0,7), [] (pgm-or-skip(mult) -> mult-inv(0))) .
red modelCheck(mult | mult-init(2,2), [] (pgm-or-skip(mult) -> mult-inv(4))) .
red modelCheck(mult | mult-init(3,2), [] (pgm-or-skip(mult) -> mult-inv(6))) .
red modelCheck(mult | mult-init(2,3), [] (pgm-or-skip(mult) -> mult-inv(6))) .
red modelCheck(mult | mult-init(3,3), [] (pgm-or-skip(mult) -> mult-inv(9))) .
```

Your task is to provide a loop invariant for the `fac` program below and to show it holds from a set of given initial states. Here is a simple factorial program written in `IMP`:

```
mod FACTORIAL is pr IMP-SEMANTICS .
  op fac      : -> Stmt [ctor] .
  op fac-init : Nat -> Memory .
  var N : Nat .
  eq fac         = while 0 < n do s := n * s ; n := (n - 1) od .
  eq fac-init(N) = {[n,N] [s,1]} .
endm
```

Proving a loop invariant holds from an initial state in this example has a very similar structure to the one in the proof that we saw above. Your proof will have the form:

```
red modelCheck(fac | fac-init(N), [] (pgm-or-skip(fac) -> < your loop invariant >) .
```

To test that your invariant is correct, please show that it holds for at least the following initial states:

```
fac | fac-init(0)
fac | fac-init(1)
fac | fac-init(2)
fac | fac-init(3)
```

What to submit: (a) your invariant specification (b) the output of the Maude LTL model checker showing that your invariant holds from the initial states given above. We have provided a stub file `fac.maude` on the course website which you can fill out in order to complete the assignment.

2. Use search or LTL model checking to verify the following properties of Peterson's algorithm:

   (a) The algorithm never deadlocks, that is, it has no terminating states.

   (b) Whenever a state reachable from the initial state is such that both `p0` and `p1` are outside their critical sections, then either `p0` and `p1` will remain outside their critical sections forever, or (at least) one of them will eventually get to its critical section.

To verify these two properties you can use the Maude modules `PETERSON` (where the initial memory `init-mem` is defined) and (for LTL model checking) `THREADED-IMP-CHECK`. They are available in the course web page as Maude code for Lecture 26, together with the two model checking examples explained in that lecture and the module `THREADED-IMP-SEMANTICS`. For your convenience, we have a stub file `threaded-imp.maude` available on the course website. You can directly append your solution to the end of the file for easier testing.