CS 476 Homework #6 Due 10:45am on Wednesday 11/20

Note: Answers to the exercises listed below and all Maude code and screenshots of tool interactions should be emailed to `meseguer@illinois.edu`.

1. **Problem** 1. Consider the following token ring mutual exclusion protocol, where processes (like in the case of dining philosophers) are arranged in a ring. Mutual exclusion is achieved by each process entering its critical section when it receives a token message with its name, and then passing on the token to the next (successor) process. This protocol is obviously *parametric* on the number $n \geq 1$ of processes. Here, the case $n = 5$ is exemplified.

```
fmod NAT/5 is protecting NAT .
   sort Nat/5 .
   op [_] : Nat -> Nat/5 .
   var I : Nat .
   ceq [I] = [I rem 5] if I > 4 .
 endfm

omod TOK-RING-5 is protecting NAT/5 .
   sort Mode .
   subsort Nat/5 < Oid .
   ops wait crit : -> Mode .
   msg tok : Nat/5 -> Msg .
   op init : -> Configuration .
   class Proc | mode : Mode .

   var I : Nat .

    eq init = tok([0])  < [0] : Proc | mode : wait >
    < [1] : Proc | mode : wait >  < [2] : Proc | mode : wait >
    < [3] : Proc | mode : wait >  < [4] : Proc | mode : wait > .

   rl [enter] : tok([I]) < [I] : Proc | mode : wait >
     => < [I] : Proc | mode : crit > .
   rl [exit] : < [I] : Proc | mode : crit >
     => < [I] : Proc | mode : wait > tok([s(I)]) .
endom
```

The main purpose of this problem is to allow you to have fun playing with Maude's LTL model checker, to become familiar with how to specify (possibly parametric) state predicates in a module (that you are supposed to define) `TOK-RING-5-PREDS` importing `TOK-RING-5`, and to specify and verify interesting properties by means of LTL formulas that you have written yourself in the LTL syntax supported by another module (that you are supposed to define) `TOK-RING-5-CHECK` that imports both `MODEL-CHECKER` and `TOK-RING-5-PREDS`. You are specifically asked to do the following:

   (a) Prove that from the initial state `init`, `TOK-RING-5` satisfies the *mutual exclusion* invariant that there are never two or more processes in their critical section at the same time. Proving this will require you to both define some state predicate or predicates, writing an LTL formula for mutual exclusion, and verifying it by model checking.

(b) Prove that from the initial state `init`, `TOK-RING-5` satisfies the *deadlock freedom* invariant. Again, proving this will require you to both define some state predicate or predicates, writing an LTL formula for deadlock freedom, and verifying it by model chcking.

(c) Write a single *non-starvation* LTL formula that says that each of the five processes will be in its critical section infinitely often. Then verify that this formula is true from the initial state `init` by model checking it.

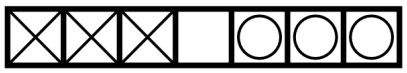
(d) **For Extra Credit** (you can gain 5 extra points on this problem if you solve (d), i.e., a total of 15 points). Write a different *non-starvation* formula (I myself call it *encore*($[i]$)) parametric on process names $[i]$, $0 \leq i \leq 4$, that describes much more precisely *when* (after how many computation steps k of the protocol) if a process $[i]$ is currently in its critical section, it will be again in its citical section. Then write a single LTL formula stating that *all* the processes $[i]$, $0 \leq i \leq 4$, enjoy this much more precise non-starvation property, and verify it from the initial state `init` using the Maude model checker.

To make your life easier, here you have a skeleton that you can fill in with your predicate and formula definitions in order to do this homework:

```
fmod NAT/5 is protecting NAT .
   sort Nat/5 .
   op [_] : Nat -> Nat/5 .
   var I : Nat .
   ceq [I] = [I rem 5] if I > 4 .
 endfm

omod TOK-RING-5 is protecting NAT/5 .
   sort Mode .
   subsort Nat/5 < Oid .
   ops wait crit : -> Mode .
   msg tok : Nat/5 -> Msg .
   op init : -> Configuration .
   class Proc | mode : Mode .

   var I : Nat .

    eq init = tok([0])  < [0] : Proc | mode : wait >
    < [1] : Proc | mode : wait >  < [2] : Proc | mode : wait >
    < [3] : Proc | mode : wait >  < [4] : Proc | mode : wait > .

   rl [enter] : tok([I]) < [I] : Proc | mode : wait >
     => < [I] : Proc | mode : crit > .
   rl [exit] : < [I] : Proc | mode : crit >
     => < [I] : Proc | mode : wait > tok([s(I)]) .
endom

in model-checker.maude

omod TOK-RING-5-PREDS is protecting TOK-RING-5 .
  extending SATISFACTION .

  subsort Configuration < State .

  vars I J : Nat .  var C : Configuration .

*** Give here your state predicate definitions.

endom

omod TOK-RING-5-CHECK is
     protecting TOK-RING-5-PREDS .
     including MODEL-CHECKER .
```

```
     var I : Nat .

*** Give here your fomula definitions.  This is not strictly
*** necessary: one can always write such formulas directly in
*** model checking commands; but is is convenient, since
*** one can give a name to a somewhat long formula and
*** then use it by that name in model checking commands.

endom
```

2. **Problem** 2. In the *Hopping Rabbits* game there are two teams of rabbits moving in opposite (left-to-right or right-to-left) directions and a single free space between them. Any rabbit next to the free space may *advance* one space forward. Furthermore, if a rabbit from one team is next to a second rabbit from the other team that is itself next to the free space, the first rabbit can *jump* over the second to occupy the free space behind its opponent. The game is parametric on the number of rabbits on each team. Let us mark each member of each team with either $\times$ or, respectively, $\bigcirc$. An initial state with three rabbits on each team looks like this:



A game will end in *success* if the rabbits, by their collective behavior, end up with all the $\bigcirc$ rabbits on the left, all the $\times$ rabbits on the right, and with the free space between them (the mirror image of the above picture).

The system module below (slightly adapted from §7.2 in the Maude book) specifies *Hopping Rabbits* in full generality. The specific initial state chosen here has 5 rabbits on each team.

```
mod RABBIT-HOP is
protecting NAT .
sorts Rabbit RabbitList .
subsort Rabbit < RabbitList .

ops x o free : -> Rabbit [ctor] .
op nil : -> RabbitList [ctor] .
op __ : RabbitList RabbitList -> RabbitList [ctor assoc id: nil] .
ops initial10 final10 : -> RabbitList .

eq initial10 = x x x x x free o o o o o .
eq solution10 = o o o o o free x x x x x .

rl [xAdvances] : x free => free x .
rl [xJumps] : x o free => free o x .
rl [oAdvances] : free o => o free .
rl [oJumps] : free x o => o x free .
rl [encore] : o o o o o free x x x x x => x x x x x free o o o o o .
endm
```

The rules of the game are specified by the first four rules. The rule `[encore]` has been added so that, if the rabbits manage to succeed in solving this puzzle, they can start all over again playing another round of the same game. The game is *non-trivial*, since it is quite easy for the rabbits to get stuck in a *deadlock* situation. In fact with 5 rabbits on each team there are 100 deadlock states (99 when we add the `[encore]` rule). So, success is a non-trivial matter. But if the two rabbit teams are smart enough to *cooperate* with each other, they may collectively solve the puzzle: it is a *win-win* situation for them.

The main purpose of this problem is to help you become familiar with the expressive power to ask useful and incisive questions about a system's behavior offered by LTL^+ formulas of the form $\mathbf{E}\,\varphi \in LTL(\Pi)^+$. You can think of any such formula as a *question* about a sytem's? behavior, namely, the question:

Is is possible for the system to exhibit a behavior of the form φ?

You are asked to formulate and get aswers to three such questions about the *Hopping Rabbits* game. Each of them should be asked by writing a formula of the form $\mathbf{E}\,\varphi \in LTL(\Pi)^+$ and getting an answer for it from the LTL model checker as a counterexample to an LTL formula of the form $\neg\varphi$, which is precisely a *proof* of $\mathbf{E}\,\varphi \in LTL(\Pi)^+$, i.e., an answer to the question $\mathbf{E}\,\varphi \in LTL(\Pi)^+$. Here are the three questions (all asked from the initial state `initial10`) that you are asked to formulate in $LTL(\Pi)^+$ and get answers to:

(a) Is it possible for the *Hopping Rabbits* system to deadlock?

(b) Is the *Hopping Rabbits* puzzle *solvable*, i.e., it is possible to get to the state `solution10` from the initial state `initial10`? (The fact that one can get an answer to this particular question (but in general not to other existence questions) using Maude's `search` command is irrelevant here: the point is to get an answer as a *proof* of an $LTL(\Pi)^+$ formula).

(c) After having shown solvability, you should ask, and get an answer to, the following further question: Is is possible for the two teams of rabbits to cooperate with each other so as to *successfully* (i.e., always getting to `solution10` from `initial10`) play the game *forever*, i.e., to play an infinite number of successful game rounds?

For each of these three questions you should:

- state explicitly what your $\mathbf{E}\,\varphi$ formula, formally asking such a question, is, and
- explain how this formula is answered by the LTL model checker in the form of a counterexample to a model checking command that proves $\mathbf{E}\,\varphi$.

Of course, you will need to first define some state predicates before you can even formulate, and get answers to, these three questions.

To make your life easier, here is a template that can help you in solving this problem:

```
mod RABBIT-HOP is
protecting NAT .
sorts Rabbit RabbitList .
subsort Rabbit < RabbitList .

ops x o free : -> Rabbit [ctor] .
op nil : -> RabbitList [ctor] .
op __ : RabbitList RabbitList -> RabbitList [ctor assoc id: nil] .
ops initial10 solution10 : -> RabbitList .

eq initial10 = x x x x x free o o o o o .
eq solution10 = o o o o o free x x x x x .

rl [xAdvances] : x free => free x .
rl [xJumps] : x o free => free o x .
rl [oAdvances] : free o => o free .
rl [oJumps] : free x o => o x free .
rl [encore] : o o o o o free x x x x x => x x x x x free o o o o o .
endm

in model-checker.maude

mod RABBIT-HOP-PREDS is
  protecting RABBIT-HOP .
  extending SATISFACTION .

  subsort RabbitList < State .

  *** Declare here your state predicates and
  *** give their equational definition.

endm
```

```
mod RABBIT-HOP-CHECK is protecting RABBIT-HOP-PREDS .
    including MODEL-CHECKER .
endm
```

3. **Problem** 3.

```
mod R&W is
  sorts Nat Config .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op <_,_> : Nat Nat -> Config [ctor] .  --- readers/writers

  vars R W : Nat .

  rl < 0, 0 > => < 0, s(0) > .
  rl < R, s(W) > => < R, W > .
  rl < R, 0 > => < s(R), 0 > .
  rl < s(R), W > => < R, W > .
endm
```

Recall that, since the above readers-and-writers protocol specification discussed in Lecture 18 (slightly modified here to avoid use of the built-in module `NAT`), is infinite-state, in Lecture 18 we were only able to perform *bounded model checking* of some invariants desired for this module, including: (i) *mutual exclusion*, and (ii) *one writer*. In this problem you are asked to verify invariants (i)–(ii) by folding narrowing-based symbolic model checking using the **backwards narrowing** method explained in Appendix 2 to Lecture 23.

Hints. Note that verifying invariants (i)–(ii) from initial state `< 0 , 0 >` by **folding (forwards) narrowing** (the usual method) is impossible, because folding narrowing search from `< 0 , 0 >` would just be performing a search equivalent to the standard `search` command in Maude (since narrowing coincides with rewriting for ground terms and it is impossible to fold a ground state into a different ground state), which can only be used effectively in a bounded search manner. However, all options are open if backwards narrowing search is used.

Caveats: (1) The Maude command to use in order to perform folding narrowing search (here in a backwards direction) is the `{fold} vu-narrow` command illustrated in Lecture 23 and also documented in the Maude 3.5 Manual. (2) Recall from pg. 13 of Lecture 23 (also explained in the Maude 3.5 Manual), that to perform narrowing-based model checking it is **absolutely necessary** that all rules in the module must be declared with the `[narrowing]` attribute. You can easily fool yourself by failing to annotate rules with the `[narrowing]` attribute, since then all `{fold} vu-narrow` search commands will return `No solution`, giving you the false impression that you have verified some property.