Problem Set 4

1. The following mutual exclusion protocol is based on the idea of a *token ring*, where processes can be in one of two modes: `wait` and `critical`, but only the process that gets the unique token in a message can enter the critical section. This protocol can be easily specified as a *parametric* system module, where the parameter n is the number of processes. See §8.2 of the Maude Book for an explanation of parameterized theories and parameterized modules. Here we have instantiated the parameter n to 5. This means that the set of object identifiers ranges ove the naturals modulo 5, defined in the functional module `NAT/5`.

   The protocol itself is specified in the system module `TOK-RING`, which imports `NAT/5`. The initial state `init` has just a message sending the token to process `[0]` and processes `[0]` to `[4]`, all in `wait` mode.

   You are asked to model check the following two properties of `TOK-RING`:

   - **Mutual Exclusion**, and
   - **Deadlock Freedom**.

   The "cool" way explained in Lecture 4 can easily be used to prove both properties.

```
fmod NAT/5 is
   protecting NAT .
   sort Nat/5 .
   op [_] : Nat -> Nat/5 .
   op _+_ : Nat/5 Nat/5 -> Nat/5 .
   op _*_ : Nat/5 Nat/5 -> Nat/5 .
   op p : Nat/5 -> Nat/5 .   *** predecessor modulo 5
   vars N M : Nat .
   ceq [N] = [N rem 5] if N >= 5 .
   eq [N] + [M] = [N + M] .
   eq [N] * [M] = [N * M] .
   ceq p([0]) = [N] if s(N) := 5 .
   ceq p([s(N)]) = [N] if N < 5 .
endfm

mod TOK-RING is
protecting NAT/5 .
sorts Mode Oid Object Msg Conf .
subsort Nat/5 < Oid .
subsort Object Msg < Conf .
op none : -> Conf [ctor] .
op _ _ : Conf Conf -> Conf [ctor assoc comm id: none] .
ops wait critical : -> Mode .
op tok : Nat/5 -> Msg .
op init : -> Conf .
op make-init : Nat/5 -> Conf .
op <_: Proc | mode:_> : Nat/5 Mode -> Object .

vars I J : Nat .  var C : Conf .

ceq init = tok([0]) make-init([I]) if s(I) := 5 .
```

```
ceq make-init([s(I)]) = < [s(I)] : Proc | mode: wait > make-init([I]) if I < 5 .
eq make-init([0]) = < [0] : Proc | mode: wait > .

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

2. Consider the following *dining philosophers* example, due to Dijksta. This example is also parametric on n. Here we have fixed the parameter $n = 4$ in a way entirely similar as how we fixed $n = 5$ for `TOK-RING`.

```
fmod NAT/4 is
   protecting NAT .
   sort Nat/4 .
   op [_] : Nat -> Nat/4 .
   op _+_ : Nat/4 Nat/4 -> Nat/4 .
   op _*_ : Nat/4 Nat/4 -> Nat/4 .
   op p : Nat/4 -> Nat/4 .  *** predecessor modulo 4
   vars N M : Nat .
   ceq [N] = [N rem 4] if N >= 4 .
   eq [N] + [M] = [N + M] .
   eq [N] * [M] = [N * M] .
   ceq p([0]) = [N] if s(N) := 4 .
   ceq p([s(N)]) = [N] if N < 4 .
endfm

mod DIN-PHIL is
   protecting NAT/4 .
   sorts Oid Cid Attribute AttributeSet Configuration Object Msg .
   sorts Phil Mode .
   subsort Nat/4 < Oid .
   subsort Attribute < AttributeSet .
   subsort Object < Configuration .
   subsort Msg < Configuration .
   subsort Phil < Cid .

   op _ _ : Configuration Configuration -> Configuration
                                                [ assoc comm id: none ] .
op _,_ : AttributeSet AttributeSet -> AttributeSet
                                                [ assoc comm id: null ] .
   op null : -> AttributeSet .
   op none : -> Configuration .
   op mode:_ : Mode -> Attribute [ gather ( & ) ] .
   op holds:_ : Configuration -> Attribute [ gather ( & ) ] .
   op <_:_|_> : Oid Cid AttributeSet -> Object .
   op Phil : -> Phil .

   ops t h e : -> Mode .   *** t = thinking, h = hungry, e = eating.
   op chop : Nat/4 Nat/4 -> Msg [comm] .
   op init : -> Configuration .
   op make-init : Nat/4 -> Configuration .

   vars N M K : Nat .   vars ATTS ATTS' ATTS''  : AttributeSet .
   var C : Configuration .

   ceq init = make-init([N]) if s(N) := 4 .
   ceq make-init([s(N)])
```

```
    = < [s(N)] : Phil | mode: t , holds: none >  make-init([N])  (chop([s(N)],[N]))
    if N < 4 .
  ceq make-init([0]) =
    < [0] : Phil | mode: t , holds: none > chop([0],[N]) if  s(N) := 4 .

  rl [t2h] : < [N] : Phil | mode: t , holds: none > =>
    < [N] : Phil | mode: h , holds: none > .
  crl [pickl] :  < [N] : Phil | mode: h , holds: none > chop([N],[M])
     => < [N] : Phil | mode: h , holds:  chop([N],[M]) > if [M] = [s(N)] .
  rl [pickr] :  < [N] : Phil | mode: h , holds: chop([N],[M]) >
    chop([N],[K]) =>
    < [N] : Phil | mode: h , holds:  chop([N],[M]) chop([N],[K]) > .
  rl [h2e] :  < [N] : Phil | mode: h , holds:  chop([N],[M])
    chop([N],[K]) > => < [N] : Phil | mode: e ,
    holds:  chop([N],[M]) chop([N],[K]) > .
  rl [e2t] :  < [N] : Phil | mode: e , holds:  chop([N],[M])
    chop([N],[K]) >  =>   chop([N],[M]) chop([N],[K])
    < [N] : Phil | mode: t , holds:  none > .
endm
```

There are four philosophers, that you can imagine eating in a circular table. Initially they are all in thinking mode (`t`), but they can go into hungry mode (`h`), and after picking the left and right chopsticks (they eat Chinese food) into eating mode (`e`), and then can return to thinking.

The identities of the philosophers are naturals modulo 4, with contiguous philosophers arranged in increasing order from left to right (but wrapping around to 0 at 4). The chopsticks are numbered, with each chopstick indicating the two philosophers next to it.

Prove, by giving appropriate search commands from the initial state `init`, the following properties:

- (contiguous mutual exclusion): it is never the case that two *contiguous* philosophers are eating simultaneously.
- (mutual non-exclusion): it is however possible for two philosophers to eat simultaneously.
- (three exclusion): it is impossible for three philosophers to eat simultaneously.
- (lack of deadlock fredom) the system can deadlock.

Again, the "cool" way of verifying all these properties can easily be used; but you can instead use the "square" way if you so desire.

Note: In trying to verify some of the above properties you may find it useful to use the built-in disequality predicate `_=/=_` that evaluates to `true` (resp. `false`) for two terms t, t' having the same or related sorts iff the canonical forms of t and t' by the module's equations are the same (resp. different) modulo the module's axioms.

3. The following example is a simplified version of Lamport's bakery protocol. This is an infinite-state protocol that achieves mutual exclusion between processes by the usual method common in bakeries and deli shops: there is a number dispenser, and customers are served in sequential order according to the number that they hold. A simple Maude specification for the case of two processes can be given as follows:

```
fmod NAT-ACU is
  sort Nat .
  ops 0 1 : -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
endfm

mod BAKERY is
  protecting NAT-ACU .
```

```
  sorts Mode BState .

  ops sleep wait crit : -> Mode [ctor] .
  op <_,_,_,_> : Mode Nat Mode Nat -> BState [ctor] .
  op initial : -> BState .

  vars P Q : Mode .
  vars X Y N M : Nat .  var S : BState .

  eq initial = < sleep, 0, sleep, 0 > .

  rl [p1_sleep] : < sleep, X, Q, Y > => < wait, Y + 1, Q, Y > .
  rl [p1_wait] : < wait, X, Q, 0 > => < crit, X, Q, 0 > .
  rl [p1_wait] : < wait, X, Q, X + Y > => < crit, X, Q, Y + X > .
  rl [p1_crit] : < crit, X, Q, Y > => < sleep, 0, Q, Y > .

  rl [p2_sleep] : < P, X, sleep, Y > => < P, X, wait, X + 1  > .
  rl [p2_wait] : < P, 0, wait, Y > => < P, 0, crit, Y > .
  rl [p2_wait] : < P, X + Y + 1, wait, Y > => < P, X + Y + 1, crit, Y > .
  rl [p2_crit] : < P, X, crit, Y > => < P, X, sleep, 0 > .
endm
```

In this module, states are represented by terms of sort `BState`, which are constructed by a 4-tuple operator `<_,_,_,_>`; the first two components describe the status of the first process (the mode it is currently in, and its priority as given by the number according to which it will be served), and the last two components the status of the second process. The rules describe how each process passes from being sleeping to waiting, from waiting to its critical section, and then back to sleeping.

We would like to verify two basic invariants about this protocol, namely:

- *mutual exclusion*, that is, the two processes are never simultaneously in their critical section, and
- *deadlock freedom*.

However, since the set of states reachable from `initial` is *infinite*, we cannot use the `search` command to *fully verify* these two invariants since, if the invariants are satisfied by `BAKERY`, Maude will never come back. However, we can *partially verify* the above two invariants up to a certain *depth bound*.

You are asked to use *bounded model checking* to partially verify up to depth 10^6 both the *mutual exclusion* and the *deadlock freedom* invariants of `BAKERY` from the initial state `initial`.