CS 476 Homework #13 Due 10:45am on 11/28

Note: Answers to the exercises listed below and all Maude code and screenshots of tool interactions should be emailed to `clarage2@illinois.edu`.

1. **Problem 1**. Any subset $U \in \mathcal{P}(\mathbb{N})$ of the natural numbers defines a Kripke structure $\mathcal{N}_U =_{def} (\mathbb{N}, s, \mathcal{N}_U)$ over the alphabet $\Pi = \{p\}$ having a single state predicate symbol p, where s denotes the successor function viewed as a transition relation $s \subset \mathbb{N} \times \mathbb{N}$, and $\mathcal{N}_U$ is the mapping:

$$\mathcal{N}_U : p \mapsto U$$

Any subset $U \in \mathcal{P}(\mathbb{N})$ of the natural numbers is either: (1) a *finite* set, or (2) an *infinite* set; and any such infinite U is either: (2.1) *cofinite*, i.e., its complement $\mathbb{N} \setminus U$ is finite, or (2.2) *infinite and not cofinite*, i.e., $\mathbb{N} \setminus U$ is infinite. Write formulas $\varphi_1, \varphi_2, \varphi_{2.1}, \varphi_{2.2} \in LTL(\{p\})$ such that for any $U \in \mathcal{P}(\mathbb{N})$:

(a) U is finite iff $\mathcal{N}_U, 0 \models_{LTL} \varphi_1$
(b) U is infinite iff $\mathcal{N}_U, 0 \models_{LTL} \varphi_2$
(c) U is cofinite iff $\mathcal{N}_U, 0 \models_{LTL} \varphi_{2.1}$
(d) U is infinite and not cofinite iff $\mathcal{N}_U, 0 \models_{LTL} \varphi_{2.2}$

and give a proof of the equivalences (a)–(d).

Give a proof also of the following four equivalences for any $U \in \mathcal{P}(\mathbb{N})$ and $n \in \mathbb{N}$:

(a) U is finite iff $\mathcal{N}_U, n \models_{LTL} \varphi_1$
(b) U is infinite iff $\mathcal{N}_U, n \models_{LTL} \varphi_2$
(c) U is cofinite iff $\mathcal{N}_U, n \models_{LTL} \varphi_{2.1}$
(d) U is infinite and not cofinite iff $\mathcal{N}_U, n \models_{LTL} \varphi_{2.2}$

Finally, prove that for *any* formula $\varphi \in LTL(\{p\})$ the following equivalence holds for any $U \in \mathcal{P}(\mathbb{N})$:

$$\mathcal{N}_U, 0 \models_{LTL} \varphi \;\; \Leftrightarrow \;\; \mathcal{N}_U, 0 \models_{LTL^+} \mathbf{E}\,\varphi.$$

2. **Problem 2: Dining Philiosophers Revisited**. This problem is a variation on the dining philosophers protocol that you were asked to analyze in Homework 10. It is now viewed *in hidsight*, with the benefit of now knowing about temporal logic. Unlike the version in Homework 10, the version that you are now presented with is (as you will prove) *deadlock free*. This is achived by adding to the dining room an adjacent *library*, were, after having dinner, a philosopher can go to read. The library has also a FIFO queue, so that philosophers can go back to the dining room following the FIFO order of their entrance in the library (this may remind you of the `QLOCK` protocol, event though the queue is used here for a different purpose).

   Here is the current specification, which you can retrieve from the course web page. Since some of the aspects that you will be asked to verify involve *fairness* issues, the technique explained in Lecture 24 of encoding some *action* information in the state is used, so that actions can be recorded. Not everything is thus recorded, but two crucial philosopher actions, namely, picking up a chopstick, and eating, are recorded in the state to facilitate stating *object fairness* properties and, more generally, various fairness properties.

```
fmod NAT/4 is
   protecting NAT .
   sort Nat/4 .
   op [_] : Nat -> Nat/4 .
   op p : Nat/4 -> Nat/4 .
   vars N M : Nat .
   ceq [N] = [N rem 4] if N >= 4 .
   eq p([0]) = [3] .
   ceq p([s(N)]) = [N] if N < 4 .
endfm

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

op [_{_}_]{_} : Configuration Queue Configuration Action -> PState [ctor] .
   *** state components: library, queue, dining room, and Action record.

op __ : Configuration Configuration -> Configuration
                                                [ctor assoc comm id: none ] .
op _,_ : AttributeSet AttributeSet -> AttributeSet
                                                [ctor assoc comm id: null ] .
op null : -> AttributeSet [ctor] .
op none : -> Configuration [ctor] .
op mode‘:_ : Mode -> Attribute [ctor gather ( & ) ] .
op holds‘:_ : Configuration -> Attribute [ctor gather ( & ) ] .
op <_:_|_> : Oid Cid AttributeSet -> Object [ctor] .
op Phil : -> Phil .
op mt : -> Queue [ctor] .
op _;_ : Queue Queue -> Queue [ctor assoc id: mt] .
ops t h e : -> Mode [ctor] .
op chop : Nat/4 Nat/4 -> Msg [comm] .
op init : -> PState .
op * : -> Action [ctor] .    *** action about which no information is recorded
ops picks eats : Nat/4 -> Action [ctor] .  *** picking and eating actions

vars N M K : Nat .    var Q : Queue .  var A : Action .
vars C C1 C2 C3 : Configuration .

eq init =
[< [0] : Phil | mode : t , holds : none > < [1] : Phil | mode : t , holds : none >
< [2] : Phil | mode : t , holds : none > < [3] : Phil | mode : t , holds : none >
{[0] ; [1] ; [2] ; [3]} chop([3],[2]) chop([2],[1]) chop([1],[0]) chop([0],[3])]{*} .

rl [t2h] : [< [N] : Phil | mode : t , holds : none > C1 {[N] ; [M] ; Q} C]{A} =>
[C1 {[M] ; Q} < [N] : Phil | mode : h , holds : none > C]{*} .

rl [pick] : [C1 {Q} < [N] : Phil | mode : h , holds : none > chop([N],[M]) C]{A} =>
[C1 {Q} < [N] : Phil | mode : h , holds : chop([N],[M]) > C]{picks([N])} .
```

```
rl [pick] :
[C1 {Q} < [N] : Phil | mode : h , holds : chop([N],[M]) > chop([N],[K]) C]{A} =>
[C1 {Q} < [N] : Phil | mode : h , holds : chop([N],[M]) chop([N],[K]) > C]{picks([N])} .

rl [h2e] :
[C1 {Q} < [N] : Phil | mode : h , holds : chop([N],[M]) chop([N],[K]) > C]{A} =>
[C1 {Q} < [N] : Phil | mode : e , holds : chop([N],[M]) chop([N],[K]) > C]{eats([N])} .

rl [e2t] :
[C1 {Q} < [N] : Phil | mode : e , holds : chop([N],[M]) chop([N],[K]) > C]{A} =>
[< [N] : Phil | mode : t , holds : none > C1{Q ; [N]} chop([N],[M]) chop([N],[K]) C]{*} .
endm
```

There first part of the problem is a *sanity check*: anything you solved in Homework 10 using `search` you should now be able to solve using LTL and LTL^+ formulas.

Prove, by giving appropriate LTL and LTL^+ formulas and model checking them from the initial state `init`, the following properties. Specifically, write LTL and LTL^+ formulas to get from the Maude LTL Model Checker answers to the following questions (and when the formula you are using is an LTL^+ formula, explain clearly what that formula is and how you get a proof of it from the Maude LTL Model Checker):

(a) (contiguous mutual exclusion): it is never the case that two *contiguous* philosophers are eating simultaneously.

(b) (mutual non-exclusion): it is however possible for two philosophers to eat simultaneously.

(c) (three exclusion): it is impossible for three philosophers to eat simultaneously.

(d) (deadlock fredom) the system is deadlock-free.

Of course, the point of LTL is that it provides a considerably richer property specification language than that of the constrained patterns used in modal logic verification with the `search` command. So, the second part of this problem is to specify and verify properties that could not be specified in Homework 10. Give appropriate LTL and LTL^+ formulas and model check them from the initial state `init` to verify the following properties, all of which have to to with *non-starvation*, i.e., with a philosopher eating infinitely often:

(a) It is always the case that at least one of the philosophers is not starved (eats infinitely often).

(b) It is however possible for some particular philosopher to not to eat infinitely often (starvation).

(c) It is possible for all philosophers to eat infinititely often during the same infinite execution.

For Extra Credit. You can get 50% extra credit on Problem 2 if you can specify a fairness assumption formula φ under which (i.e., under the assumption that that formula holds) you can verify using Maude's LTL model checker that:

- Under the assumption φ, it is always the case that all philosophers eat infinitely often.

Of course, in LTL you cannot even open your mouth unless you have previously specified the relevant *state predicates*. To facilitate your task, here is a skeleton that, after entering `NAT/4` and `DIN-PHIL` (the previous specification above) you can use to define your predicates and formulas.

```
in model-checker

mod DIN-PHIL-PREDS is
   protecting DIN-PHIL .
   including SATISFACTION .
```

```
   subsort PState < State .

vars N M K : Nat .    var Q : Queue .  var A : Action .
vars C C1 C2 C3 C4 : Configuration .

*** specify here your state predicates

endm


mod CHECK-DIN-PHIL is
  inc DIN-PHIL-PREDS .
  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .


vars N M K : Nat .    var Q : Queue .  var A : Action .
vars C C1 C2 C3 C4 : Configuration .

*** specify here your formulas

endm
```