CS 476 Comprehensive Homework
Due at 11:30 pm on Monday 12/16

Important Notes: (1) In consideration of the fact that you may be involved in various final exams, you are given a full week to solve this Comprehensive Homework. Given the very ample time you have available, except for a major, verifiable emergency, like a grave illness, there will be no extensions possible: any solutions emailed after 11:30 pm on Monday 12/16 will get 0 points. Your solutions in pdf, *which should include screenshots of all tool interactions*, as well as additional files for all Maude code (with sequeces of commands) for exercises requiring it, should be emailed to `meseguer@illinois.edu`. (2) All Maude templates for the different exercises can be obtained from the Maude files for this Comprehensive Homework, also available in the CS 476 web page.

1. Consider the following definition of binary trees with natural numbers on the leaves. Complete the module `NAT-TREE+AC` given below by defining with confluent and terminating equations the two functions called `leaves` and `inner`, that count, respectively, the number of leaf nodes of a tree, and the number of nodes in a tree that are *not* leaf nodes. For example, for the tree `((0 ^ s(0)) ^ 0) ^ s(0)` there are 4 leaf nodes (namely `0`, `s(0)`, `0`, and `s(0)`, and 3 inner nodes (corresponding to the 3 different occurrences of the `^` operator).

   Warning. Note the `set include BOOL off .` command below. You should not use any built-in operators like `if_then_else_fi` or `_==_` when defining the `leaves` and `inner` functions. This is because `NAT-TREE+AC` will later be entered into the NuITP, which does not handle built-in modules like `BOOL`.

```
set include BOOL off .

fmod PEANO+AC is sort Nat .
    op 0 : -> Nat [ctor metadata "0"] .
    op s : Nat -> Nat [ctor metadata "4"] .
    op _+_ : Nat Nat -> Nat [assoc comm metadata "8"] .
    vars N M : Nat .
    eq N + 0 = N .
    eq N + s(M) = s(N + M) .
endfm

fmod NAT-TREE+AC is protecting PEANO+AC .
    sort  Tree .  subsort Nat < Tree .
    op _^_ : Tree Tree -> Tree [ctor metadata "6"] .
    op inner : Tree -> Nat [metadata "10"] .  *** counts inner nodes
    op leaves : Tree -> Nat [metadata "12"] . *** counts tree leaves
    vars N M : Nat .  vars T1 T2 : Tree .
    *** add equations for leaves and inner here
endfm
```

   Once you have defined and tested your definitions for `leaves` and `inner` do the following, *including screenshots for each tool used* in your solutions for this homework:

   - check that `NAT-TREE+AC` (and therefore `PEANO+AC` which it contains) is RPO terminating modulo its axioms (namely, those for `+`). You can do so by giving `metadata` numerical annotations to each operator in `PEANO+AC` and `NAT-TREE+AC`, setting the module `NAT-TREE+AC` in the NuITP, and giving the command `check rpo .`

- check that `NAT-TREE+AC` is sufficiently complete using the SCC tool
- state a theorem, in the form of an equation, that gives a general law stating, for any tree `T`, the exact relation between the numbers `leaves(T)` and `inner(T)`
- give a mechanical proof of that theorem using Maude's NuITP.

2. The goal of this problem is to test your ability in specifying LTL properties of a concurrent system. Given a Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, _{\mathcal{A}})$ on a set Π of state predicates, and an initial state $a \in A$, write a temporal logic formula ψ such that:

   - $\mathcal{A}, a \models \psi$ holds iff, for any path π starting at a, a given LTL formula φ holds only on a *finite (and non-zero) number of states* of π. **Hint**: just to help you think about finding such a formula ψ (which depends of course on the given φ) you may begin by assuming that φ is just an atomic proposition p. The general case is totally similar, but considering first the case when φ is an atomic proposition p may be helpful.
   - $\mathcal{A}, a \models \psi$ holds iff, for any path π starting at a, a given LTL formula φ holds on an *infinite number of states* of π.
   - $\mathcal{A}, a \models \psi$ holds iff, for any path π starting at a, a given LTL formula φ_1 holds in all states of the form $\pi(2n)$ for any $n \in \mathbb{N}$ *and* another given LTL formulas φ_2 holds on all states of the form $\pi(1+3n)$ for any $n \in \mathbb{N}$.
   - $\mathcal{A}, a \models \psi$ holds iff, for any path π starting at a, for given LTL formulas φ_1 and φ_2, there is a *non-zero* $j \in \mathbb{N}$ such that φ_1 holds on $\pi(i)$ for $0 \leq i < j$, and φ_2 holds for any $\pi(n)$ such that $n \geq j$. **Warning**: To get this right, you might wish to think carefully about the corner cases in the semantic definition of the $\mathcal{U}$ operator.

3. **Dining Philiosophers Revisited**. This problem is a variation on the dining philosophers protocol that you were asked to analyze in Homework 5. It is now viewed *in hidsight*, with the benefit of now knowing about temporal logic. Unlike the version in Homework 5, the present version is (as you will be asked to 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 properties that you will be asked to verify involve *fairness* issues, the technique explained in Lecture 22 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 5. 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 up to 50% extra credit on this problem 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 fairness 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
```

4. Consider the following specification of the R&W-FAIR protocol, for which you are asked to used Maude's Logical Model Checker to verify that it satisfies several LTL properties. For your convenience, a template is included below to help you in doing so:

```
mod R&W-FAIR is
  sorts NzNatural Natural .
  subsorts NzNatural < Natural .
  op 0 : -> Natural [ctor] .
  op 1 : -> NzNatural [ctor] .
  op _+_ : Natural Natural -> Natural [ctor assoc comm id: 0] .
  op _+_ : NzNatural Natural -> NzNatural [ctor assoc comm id: 0] .

  sort Conf .
  op <_,_>[_|_] : Natural Natural Natural Natural -> Conf .

  vars N M K I J : Natural .
  vars N' M' K' : NzNatural .

  rl [w-in]  : < 0,0 >[0 | N] => < 0,1 >[0 | N] [narrowing] .

  rl [w-out] : < 0,1 >[0 | N] => < 0,0 >[N | 0] [narrowing] .

  rl [r-in]  : < N,0 >[M + 1 | K]
            => < N + 1,0 >[M     | K] [narrowing] .

  rl [r-out] : < N + 1,0 >[M | K]
            => < N,0 >[M | K + 1] [narrowing] .
endm
```

As explained in Lecture 28, you should enter the above module in the special version of Maude extended with code for the Maude narrowing-based logical model checker available for both Linux and MacOS at:

https://github.com/kquine/maude-model-checker/releases/tag/v3.3.1-ltlr-lmc

You can start that special version of Maude by giving the command (in my case the MacOS executable contained in my directory LTL-LMC-12-24 which contains all the relevant files for this model checker):

```
meseguer@CS-MESEGUER-MBA LTL-LMC-12-24 % ./maude-ltlr-lmc.darwin64
     \||||||||||||||||||/
   --- Welcome to Maude ---
     /||||||||||||||||||\
     Maude 3.3.1 LTLR&LMC built: Nov 26 2023 21:37:24
     Copyright 1997-2023 SRI International
   Mon Dec  9 17:10:33 2024
Maude>
```

After that, you should give the command: `load symbolic-checker` and enter the following module (enclosed in parentheses) defining the state predicates needed to verify the LTL properties mentioned below and importing the SYMBOLIC-CHECKER module. To simplify your life, a template is included below:

```
load symbolic-checker
```

```
(mod R&W-FAIR-PREDS
  is protecting  R&W-FAIR .
  extending SYMBOLIC-CHECKER .

  subsort Conf < State .

  vars N M K I J : Natural .  var N' M' K' I' J' : NzNatural .

  *** declare here each state predicate "my-pred" as follows:

  op my-pred : -> Prop .

  *** for each state predicate my-pred its semantics
  *** should be specified by confluent and sufficiently
  *** complete FVP equations of the form:
  ***
  ***  eq Conf-term |= my-pred = true [variant] .
  ***
  ***  or
  ***
  *** eq Conf-term |= my-pred = false [variant] .
  ***
  *** the [variant] attribute is essential for the
  *** tool to model check your properties

endm)
```

Recall that you must *fully define* your state predicates for both their *true* and *false* cases. That is, your equation defining such predicates should be *sufficiently complete*.

Your module `R&W-FAIR-PREDS` should have defined state predicates allowing you to specify and give commands to the Maude Logical Model Checker verifying the following LTL properties from the parametric initial state `< N,0 >[M | K]`. Recall from Lecture 28 that you can give to different model checking commands: (i) an `lmc` command, which does not use folding, and (ii) an `lfmc` command, which folds less general symbolic states into more general ones. Of course, to achieve a finite number of reachable states the `lfmc` command is the way to go. However, if you are performing bounded model checking (e.g., to find a counterexample), either the `lmc` command or the `lfmc` command may be used.

Specifically, you are asked to do the following:

(a) The mutual exclusion invariant.

(b) The one-writer invariant.

(c) An LTL symbolic model checking command to verify that the event that either somebody reads or somebody writes happens infinitely often:

(d) An LTL symbolic model checking command to perform *bounded model checking* verifying the non-starvation of writers up to depth 15 using the `lmc` command.

(e) An LTL symbolic model checking command to perform *bounded model checking* up to depth 15 allowing you to get an answer to the question of whether the non-starvation of readers always happens using the `lmc` command.

(f) **For Extra Credit**. You can earn up to 50% extra credit on this problem if, by analyzing the result obtained in (e) as well as the `R&W-FAIR` specification, you can identify the "corner case" when readers can starve and then prove that for all other states outside that "corner case" readers do not starve.

Hint. You may exclude the "corner case" by using as your initial state a disjunction of patterns that describes the *complement* of such a corner case and then give an appropriate command to the Logical LTL Model Checker to verify that from such a disjunction of patterns readers do not starve.