CS476 Last Comprehensive Homework, Due at 11:59pm on Tuesday 5/11

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 11:59pm on Tuesday 5/11 will get 0 points. Your solutions, as well as all Maude code and all screenshots for exercises requiring it, should be emailed to `cs476-staff@illinois.edu`. (2) All Maude code for the different exercises can be obtained from the Latex file for this Comprehensive Homework, also available in the CS 476 web page.

1. Solve **Ex.** 11.4 in pg. 32 of Lecture 11.

2. Consider the definition of binary trees with quoted identifiers on the leaves given in Lecture 13. Complete the module given below (available in the course web page) 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 `(('a # 'b) # 'c) # 'd` there are 4 leaf nodes (namely `'a`, `'b`, `'c`, and `'d`), and 3 inner nodes (corresponding to the 3 different occurrences of the `#` operator).

```
fmod TREE is
  protecting QID .
  sorts Natural Tree .
  subsort Qid < Tree .
  op 0 : -> Natural [ctor].
  op s : Natural -> Natural [ctor].
  op _+_ : Natural Natural -> Natural [assoc comm].
  op _#_ : Tree Tree -> Tree [ctor] .
  ops leaves inner : Tree -> Natural .
  var I : Qid .
  vars N M : Natural .
  vars T T' : Tree .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
  *** 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 it is sufficiently complete using the SCC tool
   - state a theorem, in the form of a universally quantified 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 the ITP

3. `QLOCK` is a mutual exclusion protocol proposed by K. Futatsugi, where the number of processes is unbounded. There are several versions of it. The following is a very simple specification of `QLOCK` in Maude:

```
mod QLOCK is protecting NAT .
 sorts NatMSet NatList State .
```

```
 subsorts Nat < NatMSet NatList .
 op mt : -> NatMSet [ctor] .
 op _ _ : NatMSet NatMSet -> NatMSet [ctor assoc comm id: mt] .
 op nil : -> NatList [ctor] .
 op _;_ : NatList NatList -> NatList [ctor assoc id: nil] .
 op {_<_|_|_|_>} : NatMSet NatMSet NatMSet NatMSet NatList -> State [ctor] .

 op [_] : Nat -> NatMSet .  *** set of first n numbers
 op init : Nat -> State .   *** initial state, parametric on n

 vars n i j : Nat .  vars S U W C : NatMSet .  var Q : NatList .

 eq [0] = mt .
 eq [s(n)] = n [n] .

 eq init(n) = {[n] < mt | mt | mt | nil >} .

 rl [join] : {S i < U | W | C | Q >} => {S < U i | W | C | Q >} .
 rl [n2w] : {S < U i | W | C | Q >} => {S < U | W i | C | Q ; i >} .
 rl [w2c] : {S < U | W i | C | i ; Q >} => {S < U | W | C i | i ; Q >} .
 rl [c2n] : {S < U | W | C i | i ; Q >} => {S < U i | W | C | Q >} .
 rl [exit] : {S < U i | W | C | Q >} => {S i < U | W | C | Q >} .
endm
```

Each process is identified by a natural number. The system's state (enclosed in curly braces) consists of an "outside area" (on the left), where processes not currently participating in the protocol reside, and a "protocol area" (on the right, enclosed in angle brackets), that processes can enter from the outside area by *joining* the protocol (rule `[join]`). The "protocol area" has three sets of processes (normal, waiting, and critical), plus a waiting queue, where processes willing to enter their critical section can register their name. To ensure mutual exclusion, a normal process must first register its name at the end of the waiting queue (rule `[n2w]`). When its name appears at the front of the queue, it is allowed to enter the critical section (rule `[w2c]`). When it has finished using the resources implicitly modeled by being in the critical section, it can go back to normal (rule `[c2n]`). Finally, a normal process may leave the protocol, going out to the "outside area" (rule `[exit]`).

As mentioned above, the number of processes is unbounded. However, the total number of processes in a given state never changes (although the number of such processes that at any time have actually *joined* the protocol (are inside the "protocol area") may very well change). This means that, instead of considering a single initial state, we can consider a *parametic family* of initial states `init(n)`, which has exactly `n` processes, intially all in the "outside area," with the "protocol area" intially empty. Verification of invariants by using the `search` command in Maude can be made for different choices of the parameter `n`. You are asked to verify two important invariants of `QLOCK`, namey,

- **Mutual Exclusion**, i.e., the critical section is either empty or has at most one process, and
- **Deadlock Freedom**

for the following initial states: `init(6)`, `init(7)`, and `init(8)`.

4. The asynchronous and unordered communication protocol between sender and receiver objects in the `COMM` module below is a variant of the example protocol sketched out in Lecture 15.

```
fmod NAT-LIST is protecting NAT .
 sort List .
 subsorts Nat < List .
 op nil : -> List .
 op _;_ : List List -> List [assoc id: nil] .
 op length : List -> Nat .
```

```
 var L : List .
 var N : Nat .
 eq length(nil) = 0 .
 eq length(N ; L) = s(length(L)) .
endfm

mod COMM is protecting NAT-LIST . protecting  QID .
 sorts Oid Class Object Msg Msgs Att Atts Configuration State .
 subsort Qid < Oid .
 subsort Att < Atts .     *** Atts is set of attribute-value pairs
 subsort Object < Configuration .
 subsorts Msg < Msgs < Configuration .
 op none : -> Msgs [ctor] .
 op __ : Configuration Configuration -> Configuration
                       [ctor config assoc comm id: none] .
 op __ : Msgs Msgs -> Msgs
                       [ctor config assoc comm id: none] .
 op null : -> Atts .
 op _,_ : Atts Atts -> Atts [ctor assoc comm id: null] .
 op buff:_ : List -> Att [ctor] .
 op snd:_ : Oid -> Att [ctor] .
 op rec:_ : Oid -> Att [ctor] .
 op cnt:_ : Nat -> Att [ctor] .
 op ack-w:_ : Bool -> Att [ctor] .
 ops Sender Receiver : -> Class [ctor] .
 op <_:_|_> : Oid Class Atts -> Object [ctor] .
 msg to_from_val_cnt_ : Oid Oid Nat Nat -> Msg [ctor] .
 msg to_from_ack_ : Oid Oid Nat -> Msg [ctor] .
 op {_} : Configuration -> State [ctor] .
 op init : Oid Oid List -> State .

 vars N M : Nat . var L : List .  vars A B : Oid .  var C : Configuration .

 rl [snd] : {< A : Sender | buff: (N ; L), rec: B, cnt: M, ack-w: false > C}
   =>
    {(to B from A val N cnt M)
    < A : Sender | buff: L, rec: B, cnt: M, ack-w: true > C} .

 rl [rec] : {< B : Receiver | buff: L, snd: A, cnt: M >
      (to B from A val N cnt M) C}
    =>
      {< B : Receiver | buff: (L ; N), snd: A, cnt: s(M) >
      (to A from B ack M) C} .

 rl [ack-rec] : {< A : Sender | buff: L, rec: B, cnt: M, ack-w: true >
    (to A from B ack M) C}
    =>
    {< A : Sender | buff: L, rec: B, cnt: s(M), ack-w: false > C} .

 eq init(A,B,L) = {< A : Sender | buff: L, rec: B, cnt: 0, ack-w: false >
                   < B : Receiver | buff: nil, snd: A, cnt: 0 >} .
endm

rew init('a,'b,(1 ; 2 ; 3)) .
```

```
rew init('a,'b,(1 ; 2 ; 3 ; 4)) .

rew init('a,'b,(1 ; 2 ; 3 ; 4 ; 5)) .
```

The only differences are: (1) the buffers now are not separate objects: they are attributes of sender and receiver objects; (2) the sender, before sending the next item in its buffer, awaits until after receiving an `ack` from the receiver for the previous item; and (3) to facilitate the definition of state predicates, the entire configuration of objects and messages in enclosed in curly braces as a term of sort `State`. You may want to run a few tests cases. For example, those given in the rewrite commands after the `COMM` module, to get a better feeling for how this protocol works. If you wish to see a detailed trace of the executions, you can type in Maude:

```
Maude> set trace on .
```

The point about this exercise is to make you familiar with *parametric* state predicates and their model checking verification. The key idea is that, as illustrated in Problem 3 above by the `QLOCK` example, we may be interested in verifying properties, not for a single initial state `init`, but for a *parametric family of initial states*, defined using and operator:

```
op init : S1 ... Sn -> State .
```

and giving appropriate equations, were the sorts `S1 ... Sn` are its *parameter sorts*. But then, some property `P`, which we want to verify is an invariant for all these initial states, may also be itself *parametric* in the exact same sense, i.e., be a Boolean-valued state predicate of the form:

```
op P : State S1 ... Sn -> Bool .
```

If `u1 ... un` are ground terms of the parameter sorts `S1 ... Sn`, then we can model check property `P` as an invariant from the initial state `init(u1 ... un)` by failing to get any solutions for the search command:

```
search  init(u1 ... un) =>* X:State s.t.  P(X:State,u1,...,un) =/= true .
```

Specifically, you asked to correctly define an (unparametric) state predicate [which is not an invariant], and two parametric state predicates [corresponding to two invariants], whose operator declarations are given in the `COMM-PREDS` module below. The state predicates are the following:

- **Enabled**: a state is not a deadlock state and can therefore transition to some other state.

- **In-Order** (Communication). Any initial state of the form `init(A,B,L)` satisfies the parametric state predicate `In-Order(X:State,A,B,L)` (which you are asked to define) as an *invariant*, where `In-Order(X:State,A,B,L)` essentially states that if in state `X:State` `L1` is the list in the buffer of the receiver `B`, then the list `L1` is a *prefix sublist* of the list `L`. This means that the protocol achieves *in-order-communication* in spite of being asynchronous.

- **Succcess** (of the Communication). Any initial state of the form `init(A,B,L)` satisfies the parametric state predicate `Success(X:State,A,B,L)` (which you are asked to define) as an *invariant*, where `Success(X:State,A,B,L)` essentially states that either `X:State` is **Enabled**, or `B` holds `L` in its buffer. Since this protocol is *terminating*, this invariant ensures that, not only it delivers data in order, but it does indeed *succeed* in always delivering *all the data* initially stored in the sender's buffer.

You can define these predicates by giving your definition [and that of any auxiliary functions you may need] in the following module importing `COMM`. You can make use the the `[owise]` feature in Maude to define the `false` case for each predicate. **Hint**. In order to define the `Success(X:State,A,B,L)` predicate more easily, you may find it useful to define as well an auxiliary parametric predicate.

```
mod COMM-PREDS is
 protecting COMM .

 op Enabled : State -> Bool .
 ops In-Order Success : State Oid Oid List -> Bool .

 var MS : Msgs . var C : Configuration .  vars L L1 L2 : List .
 vars A B : Oid .  vars N M : Nat .  var T : Bool .  var S : State .

*** include here your equational definition of Enabled(S)

*** include here your equational definition of In-Order(S,A,B,L)

*** include here your equational definition of Success(S,A,B,L)

endm
```

After you have defined [hopefully correctly] the above state predicates, you are asked to verify in Maude the two parametric invariants `In-Order(S,A,B,L)` and `Success(S,A,B,L)` for three initial states of the form `init(A,B,L)` where `A` is `'a`, `B` is `'b`, and `L` is, respectively: `(1 ; 2 ; 3)`, `(1 ; 2 ; 3 ; 4)`, and `(1 ; 2 ; 3 ; 4 ; 5)`.

The entire *template* to be filled in, containing all the modules for this problem is, therefore:

```
fmod NAT-LIST is protecting NAT .
 sort List .
 subsorts Nat < List .
 op nil : -> List .
 op _;_ : List List -> List [assoc id: nil] .
 op length : List -> Nat .
 var L : List .
 var N : Nat .
 eq length(nil) = 0 .
 eq length(N ; L) = s(length(L)) .
endfm

mod COMM is protecting NAT-LIST . protecting  QID .
 sorts Oid Class Object Msg Msgs Att Atts Configuration State .
 subsort Qid < Oid .
 subsort Att < Atts .     *** Atts is set of attribute-value pairs
 subsort Object < Configuration .
 subsorts Msg < Msgs < Configuration .
 op none : -> Msgs [ctor] .
 op __ : Configuration Configuration -> Configuration
                       [ctor config assoc comm id: none] .
 op __ : Msgs Msgs -> Msgs
                       [ctor config assoc comm id: none] .
 op null : -> Atts .
 op _,_ : Atts Atts -> Atts [ctor assoc comm id: null] .
 op buff:_ : List -> Att [ctor] .
 op snd:_ : Oid -> Att [ctor] .
 op rec:_ : Oid -> Att [ctor] .
 op cnt:_ : Nat -> Att [ctor] .
 op ack-w:_ : Bool -> Att [ctor] .
 ops Sender Receiver : -> Class [ctor] .
 op <_:_|_> : Oid Class Atts -> Object [ctor] .
```

```
 msg to_from_val_cnt_ : Oid Oid Nat Nat -> Msg [ctor] .
 msg to_from_ack_ : Oid Oid Nat -> Msg [ctor] .
 op {_} : Configuration -> State [ctor] .
 op init : Oid Oid List -> State .

 vars N M : Nat . var L : List .  vars A B : Oid .  var C : Configuration .

 rl [snd] : {< A : Sender | buff: (N ; L), rec: B, cnt: M, ack-w: false > C}
   =>
    {(to B from A val N cnt M)
    < A : Sender | buff: L, rec: B, cnt: M, ack-w: true > C} .

 rl [rec] : {< B : Receiver | buff: L, snd: A, cnt: M >
      (to B from A val N cnt M) C}
    =>
      {< B : Receiver | buff: (L ; N), snd: A, cnt: s(M) >
      (to A from B ack M) C} .

 rl [ack-rec] : {< A : Sender | buff: L, rec: B, cnt: M, ack-w: true >
    (to A from B ack M) C}
    =>
    {< A : Sender | buff: L, rec: B, cnt: s(M), ack-w: false > C} .

 eq init(A,B,L) = {< A : Sender | buff: L, rec: B, cnt: 0, ack-w: false >
                   < B : Receiver | buff: nil, snd: A, cnt: 0 >} .
endm

rew init('a,'b,(1 ; 2 ; 3)) .

rew init('a,'b,(1 ; 2 ; 3 ; 4)) .

rew init('a,'b,(1 ; 2 ; 3 ; 4 ; 5)) .

mod COMM-PREDS is
 protecting COMM .

 op Enabled : State -> Bool .
 ops In-Order Success : State Oid Oid List -> Bool .

 var MS : Msgs . var C : Configuration .  vars L L1 L2 : List .
 vars A B : Oid .  vars N M : Nat .  var T : Bool .  var S : State .

*** include here your equational definition of Enabled(S)

*** include here your equational definition of In-Order(S,A,B,L)

*** include here your equational definition of Success(S,A,B,L)

endm
```

5. 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 is as follows:

```
set include BOOL off .

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 : Nat .

  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`.