CS476 Last Comprehensive Homework
Due at 11:59pm on Monday 12/12

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 Monday 12/12 will get 0 points. Your solutions, as well as all Maude code for exercises requiring it, should be emailed to `nishant2@illinois.edu`. In addition, your screenshots for interactions with tools *should be present in the same pdf containing your answers to the homework's problems.* (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. Let $h : \mathbb{A} \longrightarrow \mathbb{B}$ be a Σ-isomorphism, and $u = v$ a Σ-equation. Prove that

$$\mathbb{A} \models u = v \quad \Leftrightarrow \quad \mathbb{B} \models u = v.$$

For Extra Credit. You can earn 10 more points (out of the total of 10 for this exercise) if you also prove that the above equivalence generalizes to one of the form:

$$\mathbb{A} \models \varphi \quad \Leftrightarrow \quad \mathbb{B} \models \varphi.$$

for φ a *quantifier-free* Σ-formula (recall the Appendix on First-Order Logic to Lecture 14).

2. Consider the definition of binary trees with natural numbers on the leaves given in Lecture 16. 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).

```
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` is confluent using the Church-Rosser Checker
- check that `NAT-TREE+AC` is terminating using the MTA with the RPO order specified by the `metadata` annotations
- 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.

3. The asynchronous and unordered communication protocol between sender and receiver objects in the `COMM` module below is a variant of the example protocol in Lecture 17.

```
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 increase your familiarity with *parametric* state predicates and their model checking verification. Parametric state predicates are very useful for both verifying invariants with the `search` command (the case in this exercise), and to perform LTL model checking of parametric properties. The key idea is that we may be interested in verifying properties *that depend on the initial state*; but not for a single initial state `init`, but, instead, 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 are 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 the protocol 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.

```
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
```

4. The following example is a simplified version of Lamport's bakery protocol, of which several versions have been presented in CS 476 lectures. 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.

The problem, of course, is that, even in this simpler version, `BAKERY` is infinite-state.

We would like to verify three LTL properties about this protocol, namely:

- *mutual exclusion*, that is, the two processes are never simultaneously in their critical mode
- *non-starvation of process 1.* If process 1 is in its waiting mode infinitely often, then it will be in its critical mode infinitely often
- *non-starvation of process 2.* If process 2 is in its waiting mode infinitely often, then it will be in its critical mode infinitely often.

Since the set of states reachable from `initial` is *infinite*, we cannot use Maude's explicit-state LTL model checker to verify these properties. We can, however, use Maude's *Logical LTL Model Checker* to verify that these three properties hold from an *even more general symbolic initial state of the form*:

```
 < sleep , X:Nat , sleep, X:Nat >
```

To perform this verification, you first need to equationally define state predicates `1wait`, `2wait`, `1crit` and `2crit`, corresponding to the case when process 1 (resp. process 2) is in its `wait`, resp. `crit` mode. Of course, the equations D defining these predicates should be FVP. But if the righthand sides of all equations are either `true` or `false` and you have specified the equations D correctly, so that they are confluent ad sufficiently complete, they *will* be FVP.

To facilitate your work and the use of Maude's Logical LTL Model Checker you just need to: (1) fill in the remaining equations to *fully define* the above for predicates in an FVP manner in the template below; (2) enter the full template into the special version of Maude running the Maude's Logical LTL Model Checker; and (3) give three commands of the form:

```
( lfmc < sleep , X:Nat , sleep, X:Nat > |= formula1 ) .)

( lfmc < sleep , X:Nat , sleep, X:Nat > |= formula2 ) .)

( lfmc < sleep , X:Nat , sleep, X:Nat > |= formula3 ) .)
```

where `formulai` expresses in LTL property `i` for `i` = 1,2,3.

```
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


load symbolic-checker

( mod BAKERY-PREDS is
  extending SYMBOLIC-CHECKER .
  protecting BAKERY .

  subsort BState < State .
  ops 1wait 2wait 1crit 2crit : -> Prop [ctor] .

  vars P Q : Mode .
  vars X Y : Nat .

  eq < wait, X, Q, Y > |= 1wait = true  [variant] .

  *** give here the remaining equations fully defining
  *** 1wait, 2wait, 1crit, and 2crit
  *** remember to include the [variant] attribute in all equations

endm )
```

Detailed instructions on how to run the Maude LTL Logical Model Checker can be found in the CS 476 web page. Furthermore, a tool tutorial/manual can be found in:

`https://maude.cs.uiuc.edu/tools/lmc/manual.pdf`

The tutorial uses various examples available in:

`https://maude.cs.uiuc.edu/tools/lmc/`