CS476 Last Comprehensive Homework
Due at 11:59 am on Thursday 12/14

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 11:59 am on Thursday 12/14 will get 0 points. Your solutions, as well as all Maude code for exercises requiring it, should be emailed to `clarage2@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 maude files 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 up to 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 15).

2. This problem has two aspects. One the one hand, it is an exercise in proving program equivalence. On the other hand, you can learn something important from doing it; because it raises and answers the question:

   What is the gold standard for specifying natural number arithmetic?

   There are of course many "ad-hoc" ways of specifying natural number addition and multiplication: with Peano notation, counting with one's fingers, in decimal notation, in binary notation, and so on. They are "ad hoc," because they do not spell out in a simple and clear way what *kind* of algebra the natural numbers with addition and mutiplication are.

   The simplest, most convincing answer to the above questions is:

   The natural numbers are the initial algebra of the theory of commutative semirings.

   The theory of commutative semirings is a subtheory of the theory of commutative rings that drops the requirement that every element has an inverse under addition but keeps all other axioms of the theory of commutative rings. We can specify it as a functional *theory* in Maude as follows:

```
fth COMM-SEMIRING is
    sort Semiring .
    op 0 : -> Semiring .
    op 1 : -> Semiring .
    op _+_ : Semiring Semiring -> Semiring [assoc comm] .
    op _*_ : Semiring Semiring -> Semiring [assoc comm] .

    vars X Y Z : Semiring .
```

```
    eq X + 0 = X .
    eq X * 0 = 0 .
    eq X * 1 = X .
    eq X * (Y + Z) = (X * Y) + (X * Z) .
endfth
```

Changing the sort name `Semiring` to `Natural` for readability reasons (something mathematically immaterial) we just get a definition of the natural numbers essentially by replacing `fth` by `fmod` as follows (an RPO order has also been declared for theorem proving purposes):

```
set include BOOL off .

fmod NAT-SEMIRING is
    sort Natural .
    op 0 : -> Natural [ctor metadata "1"] .
    op 1 : -> Natural [ctor metadata "2"] .
    op _+_ : Natural Natural -> Natural [ctor metadata "3" assoc comm] .
    op _*_ : Natural Natural -> Natural [metadata "4" assoc comm] .

    vars X Y Z : Natural .

    eq X + 0 = X .
    eq X * 0 = 0 .
    eq X * 1 = X .
    eq X * (Y + Z) = (X * Y) + (X * Z) .
endfm
```

What you are asked to do in this problem is to prove, using the NuITP, that the above functional program `NAT-SEMIRING` is *equivalent* to the following functional program `NAT-AC`, which defines multiplication by the obvious recursive equations:

```
set include BOOL off .

fmod NAT-AC is
    sort Natural .
    op 0 : -> Natural [ctor metadata "1"] .
    op 1 : -> Natural [ctor metadata "2"] .
    op _+_ : Natural Natural -> Natural [ctor metadata "3" assoc comm] .
    op _*_ : Natural Natural -> Natural [metadata "4" assoc comm] .

    vars X Y Z : Natural .

    eq X + 0 = X .
    eq X * 0 = 0 .
    eq X * 1 = X .
    eq X * (Y + 1) = X + (X * Y) .
endfm
```

You are asked, not only to give a proof of equivalence between `NAT-SEMIRING` and `NAT-AC` using the NuITP, but also to explain *why* your proof shows them equivalent; that is, using what theorem proved in what lecture of CS 476, your proof using the NuITP shows these two programs equivalent.

Remark. If you are intellectually curious, this problem may have raised in your mind a very similar question:

> *What is the gold standard for specifying integer arithmetic?*

and you may have guessed the answer:

The integers are the initial algebra of the theory of commutative rings.

Declarative programming is, indeed, mathematical modeling.

3. 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 Natural -> Conf .

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

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

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

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

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

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' : 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 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, and that you *must use the special version of Maude* available for both Linux and MacOS at:

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

Then you:

- enter `R&W-FAIR` into this special version of Maude.
- load the file `symbolic-checker` that comes with the distribution of the special Maude version, and
- load, *enclosed in parentheses*, the module `R&W-FAIR-PREDS` for which the template has been provided above.

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 `[K + N + M]< N,0 >[M | K]`

(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 10 more points (out of the total of 10 for this exercise) if by analyzing the result obtained in (e) as well as the `R&W-FAIR` specification to identify the "corner case" when readers get starved, you can define a predicate `P` on states avoiding such a corner case so that you can prove a conditional property of the form:

```
P -> readers-do-not-starve
```

by giving an appropriate command to the Logical LTL Model Checker to verify this conditional property.

4. In this last problem you are asked to use Maude's Deductive Model Checker **DM-Check**, available at:

`https://safe-tools.dsic.upv.es/dmc/`

to specify and verify several properties about the following `TOKEN-MUTEX` protocol, which is parametric on the number `N` of processes. Since to specify both the parametric initial state and other properties some constraints for the *constrained patterns* involved in describing such properties some *auxiliary functions* are needed, the `TOKEN-MUTEX-AUX` module includes the definition of those auxiliary functions:

```
mod TOKEN-MUTEX is
  sort Number .
  op 0 : -> Number [ctor] .
  op s : -> Number [ctor] .
  op __ : Number Number -> Number [ctor comm assoc id: 0] .
```

```
  sorts Mode ModeWait .
  subsorts ModeWait < Mode .
  op crit : -> Mode [ctor] .
  op wait : -> ModeWait [ctor] .

  sorts Proc Token ProcWait ConfWait Conf .
  subsort Token < Conf .
  subsorts ProcWait < Proc ConfWait < Conf .
  op <_> : Number -> Token [ctor] .
  op [_,_] : Number Mode -> Proc [ctor] .
  op [_,_] : Number ModeWait -> ProcWait [ctor] .
  op none : -> ConfWait [ctor] .
  op __ : Conf Conf -> Conf [ctor comm assoc id: none] .
  op __ : ConfWait ConfWait -> ConfWait [ctor ditto] .


  sort Top .
  op {_|_} : Number Conf -> Top .

  vars N M : Number . var CF : Conf .

  rl [enter] : {  s N M |   < M > [M, wait] CF}
            => {  s N M |         [M, crit] CF} [narrowing] .
  rl [exit1] : {s s N M |         [M, crit] CF}
            => {s s N M | < s M > [M, wait] CF} [narrowing] .
  rl [exit2] : {    s M |         [M, crit] CF}
            => {    s M |   < 0 > [M, wait] CF} [narrowing] .
endm


mod TOKEN-MUTEX-AUX is protecting TOKEN-MUTEX .
  ops noToken allWait : Conf -> Bool .

  vars N M I J K : Number . vars CF CF1 CF2 : Conf .

  eq noToken(none) = true .
  eq noToken(< M > CF) = false .
  eq noToken([N, wait] CF) = noToken(CF) .
  eq noToken([N, crit] CF) = noToken(CF) .

  eq allWait(none) = true .
  eq allWait(< M > CF) = allWait(CF) .
  eq allWait([N, wait] CF) = allWait(CF) .
  eq allWait([N, crit] CF) = false .
endm
```

The parametric initial state can be specified by the following constrained pattern:

```
  {N | < 0 > CF } | allWait(CF) = true /\ noToken(CF) = true
```

that is, the token is sent to process `0` and all processes in `CF` are waiting. Note that this is a quite loose specification of the initial states, since it can contain "junk" waiting processes whose identifier `M` is greater or equal to the bound `N` and therefore will *never* get the token. However, for the *safety* properties that we wish to verify this over-specification of the initial states will make life easier and will suffice.

You are asked to:

(a) Specify as a disjunction of constrained patterns an *inductive invariant* from the symbolic initial state `{N | < 0 > CF } | allWait(CF) = true /\ noToken(CF) = true`

(b) Verify using the `subsumed-by` command that your conjectured inductive invariant does indeed subsume the symbolic initial state.

(c) Verify using the `check-inv` command that your conjectured inductive invariant is indeed an inductive invariant.

(d) Verify in a **Negative** way that `TOKEN-MUTEX` satisfies the invariant of *mutual exclusion* from the symbolic initial state `{N | < 0 > CF } | allWait(CF) = true /\ noToken(CF) = true`

Note. Recall from Lectures 28 and 29 that the negative verification of invariants uses *unification* in Maude (in the module `TOKEN-MUTEX`) to prove that the intersection between the inductive invariant and the pattern (or patterns) specifying the *complement* of the invariant to be proved is *empty*. However, since in this problem the patterns involved are *constrained patters*, it is perfectly possible that some unifiers will exists. In such a case you will need to show that the instance of the corresponding *constraints* by such unifiers are *unsatisfiable*, so that the intersection is indeed empty.