CS476 Last Comprehensive Homework, Due at 11am on Monday 5/6

Note: You should email by the above deadline both a pdf version of your answers to all questions and the Maude code needed for Problems 3-4 to `fanyang6@illinois.edu`. The time of 11am is a **hard deadline**.
All tools and code necessary for completing the exam are available from the course webpage.
For reasonable tool-related questions/difficulties regarding Problems 3 and 4 below you can contact Stephen Skeirik (`skeirik2@illinois.edu`).

1. Solve **Ex.** 10.4 In Lecture 10.

2. Fairness assumptions are very important to prove *liveness* properties about a system. Many liveness properties do not hold for *all behaviors* of the system, but only for *reasonable behaviors*, namely, fair ones.

   A very common notion of fairness is *transition fairness*, which for a rewrite theory can be understood as *rule fairness*, namely, for all rules with a common label l, rule fairness is the LTL formula $fair.l$, defined by the equality:

   $$fair.l = (\Box\Diamond\, enabled.l) \rightarrow (\Box\Diamond\, taken.l)$$

   which in plain English says that if l is infinitely often enabled to be fired, then l is infinitely often taken (fired).

   You have already seen examples of how the *enabled* state predicate can be defined in Lecture 18 (for all rules, but obviously this can be specialized to each rule label). Defining the *taken.l* state predicate is slightly more tricky, since just looking at a state we may have no idea about which was the last rule applied. But it can be also easily done by *tagging* the state with the last rule label used. You saw an example of this tagging in the semantics of the `THREADED-IMP` language, and of how to express a fairness assumption (that both threads execute infinitely often) by using tagging in the model checking of liveness properties (non-starvation) for Peterson's algorithm in that lecture. There, what was tagged on the state was not quite the rule label, but the thread identifier of the last thread computing; but the idea is similar: we can instead tag the state with the label of the rule producing it.

   All this was by way of introduction to make clear that you *can* express rule fairness in your specifications. From now on let us assume you have done that. Let us come to the point about this problem: there is another very simple and closely related property for a rule l, namely the "leads to" property:

   $$enabled.l \rightsquigarrow taken.l$$

   Of course, the above "leads to" property is also closely related to the LTL formula:

   $$enabled.l \rightarrow \Diamond taken.l$$

   You are asked to answer the following questions about the relationships between: (a) the property $fair.l$, (b) the $enabled.l \rightsquigarrow taken.l$ property, and (c) the property $enabled.l \rightarrow \Diamond taken.l$: (i) are any of them semantically equivalent formulas? (ii) are some of them different? (iii) if different, does satisfaction of any of them imply satisfaction of some of the others? (iv) are any two of these properties *independent*, in the sense that none of the two implies the other?

   You should give a *mathematical proof* of your answers to (i)–(iv) in terms of the *semantics* of LTL. Some of your answers may take the form of *counterexamples*.

   Hint: Since LTL semantics is universally quantified over all the paths of a Kripke structure from an initial state, it may sometimes be useful to reason about what happens on a sigle path. For instance, to show that a given formula holds on that path but another formula does not; or that if one formula holds on the path, then another formula must also hold.

3. The Pipeline Java program below is available with this homework:

```
class Main {
    static public void main (String argv[]) {
        Connector c1, c2, c3;
        c1 = new Connector();
        c2 = new Connector();
        c3 = new Connector();

        Stage s1, s2;
        Listener l;
        s1 = new Stage(1, c1, c2);
        s2 = new Stage(2, c2, c3);
        l  = new Listener(c3);

        s1.start();
        s2.start();
        l.start();

        for (int i=1; i<2; i=i+1) c1.add(i);
        c1.stop();
    }
}

class Connector {
    public int queue = -1;
    public synchronized int take(){
        int value;
        while ( queue < 0 )
            try {wait();} catch (InterruptedException ex) {}
        value = queue; queue = -1; return value;
    }
    public synchronized void add(int o) { queue = o; notifyAll(); }
    public synchronized void stop(){ queue = 0; notifyAll(); }
}

class Stage extends Thread {
    int id; Connector c1, c2;
    int stop = -1;
    public Stage(int i, Connector a1, Connector a2)
        { id = i; c1 = a1; c2 = a2; }
    public void run() {
        int tmp = -1;
        while (tmp != 0)
            if ((tmp=c1.take()) != 0){
                c2.add(tmp+1);
            }
        stop = 0;
        c2.stop();
    }
}

class Listener extends Thread {
    Connector c;
    int stop = -1;
    public Listener(Connector con) { this.c = con; }
```

```
    public void run(){
        int tmp = -1;
        while (tmp != 0)
            if ((tmp=c.take()) != 0);
        stop = 0;
        System.out.println("Listener stop.");
    }
}
```

It simulates a pipeline architecture. A desired property for this program is the propagation of termination: if the first stage stops, the final listener should stop eventually. Please use JavaFAN to verify the correctness of this program w.r.t. the propagation of termination property.

Hint: a special field, stop, is added to the program to indicate the stoppage of the stages.

Note. The JavaFAN tool is available on the course webpage. If you have difficulty or some reasonable questions regarding the use JavaFAN, you can send email to Stephen Skeirik (`skeirik2@illinois.edu`).

4. The module `SIMPLE-COMM-PROTOCOL` below models a simple communication protocol involving two participants: a sender and a receiver. The sender and receiver are linked by an error-free channel that can transport one datagram at a time. The protocol is defined by the `send-1`, `send-2`, and `recv` rewrite rules shown below. The protocol has a very simple semantics. Each datagram is modelled as a term of sort `Data`. The list of datagrams to be sent and the list of datagrams already received are both modelled as terms of sort DataList. The channel state is modelled by a term of sort `MaybeData` which represents either a single datagram or nothing.

   Protocol Correctness Property: Any reasonable communication protocol needs to satisfy a basic correctness property, namely, starting from an initial state in which the sender wants to send a list of datagrams, we always should reach a state where the receiver has received that same list of datagrams.

   Problem Overview: In this problem, your task is to verify that `SIMPLE-COMM-PROTOCOL` satisfies the above correctness property using reachability logic and the Maude reachability logic tool.

   Methodology: The correctness property above cannot be verified by model checking alone, since the protocol has *an infinite number of potential initial states*. Thus, we need the extra power of reachability logic to verify it. Recall that, in reachability logic, when we want to capture looping behavior, we need to use the Axiom inference rule. Of course, in order to use the Axiom inference rule, we need a reachability formula that can be used as a loop axiom, i.e. that captures the "looping behavior" inherent in our system. Such a reachability formula has the general structure:

   ```
   (IntermediatePattern | IntermediateConstraint) => (PostPattern | PostConstraint)
   ```

   where:

   - the constrained term `(IntermediatePattern | IntermediateConstraint)` captures *all possible* intermediate states; and
   - the constrained term `(PostPattern | PostConstraint)` captures the state corresponds to *finishing* the loop execution starting from the intermediate states described by:
     `(IntermediatePattern | IntermediateConstraint)`.

   Note that the correctness property as given above *cannot* coincide with the "looping behavior" Axiom just described because: (i) the Axiom's preconditions must capture all *intermediate states*; and (ii) the original Correctness Property's precondition must capture only *intial states*.

   Thus, to verify the correctness property, we need to *simultaneously prove two* reachability formulas: one specifying the Correctness Property and another specifying the "looping behavior" Axiom inherent in `SIMPLE-COMM-PROTOCOL`.

   Detailed Problem Description: For full credit, you need to fill out the template below (BUT SEE THE ADDITIONAL REQUIREMENT BELOW). For item number `N`, it occurs in the template as `<#N>`. In particular, you need to fill out the following four items:

(a) terminating state pattern – you must find a pattern that corresponds to all possible terminating states for this system

(b) "looping" reachability formula – you must find a reachability formula whose:

- precondition corresponds to an *arbitrary intermediate state* of the protocol
- postcondition corresponds to the final state of the protocol given the intermediate state pattern above

(c) correctness reachability formula – you must find a reachability formula whose:

- precondition corresponds to an *initial* state of the protocol, i.e. no datagrams sent yet
- postcondition corresponds to a *correct* final state of protocol, i.e. all datagrams sent by the sender are recieved in the correct order

(d) proof commands – you must use applications of either the auto command or the case command to the complete the proof.

ADDITIONAL REQUIREMENT (Postcondition Shared Variables Restricted to Constraint):

Given a reachability formula `R` of the form `A => B`, let the shared variables of `R` denote the set of variables shared between patterns `A` and `B`.

Given a constrained term `(T | C)`, we call `T` the term part and `C` the constraint part.

For technical reasons, for ALL reachability formulas you write in this problem, any SHARED variables appearing in the postcondition MUST occur ONLY in the constraint part

(for the precondition, no such restrictions are needed).

Of course, semantically, this is no problem, since we can always *abstract* any shared variables out of the term part and into the constraint, e.g.

given reachability formula:

```
g(V:Nat) | (p(V:Nat)) = (r(Q:Nat)) => f(V:Nat,W:Nat) | true
```

we see that the variable `V` is shared AND it occurs in the term part of the postcondition (what we need to avoid); thus, we transform it into:

```
g(V:Nat) | (p(V:Nat)) = (r(Q:Nat)) => f(Z:Nat,W:Nat) | (Z:Nat) = (V:Nat)
```

where `Z` is a fresh variable that is *not* shared.

```
mod SIMPLE-COMM-PROTOCOL is
  sorts Data MaybeData NeDataList DataList .
  subsorts Data < MaybeData NeDataList < DataList .
  ops a b c : -> Data [ctor] .
  op none : -> MaybeData [ctor] .
  op __ : DataList   DataList   -> DataList   [assoc] .
  op __ : NeDataList NeDataList -> NeDataList [ctor ditto] .

  sort Config State .

  op _|_|_ : DataList MaybeData DataList -> Config [ctor] .

  op {_} : Config -> State [ctor] .

   vars L L1 L2 : DataList .
   var  NL : NeDataList .
   var  D : Data .

  eq L1 none      = L1    [variant] .
```

```
  eq none L1      = L1    [variant] .
  eq (L1 none) L2 = L1 L2 [variant] .

  rl [send-1] : {D    | none | L2} => {none | D    | L2  } .
  rl [send-2] : {D NL | none | L2} => {NL   | D    | L2  } .
  rl [recv]   : {  L1 | D    | L2} => {L1   | none | L2 D} .
endm

load ../../rltool.maude


(select SIMPLE-COMM-PROTOCOL .)

(use tool varsat for validity on SIMPLE-COMM-PROTOCOL .)

(def-term-set <#1> .)

(add-goal loop    : <#2> .)

(add-goal correct : <#3> .)

(start-proof .)

(on 2 use strat loop .)

<#4>
```