Note: You should hand by the above deadline a hardcopy in latex, or similar formatting language, to Ms. Donna Coleman, my administrative assistant, who is in office SC 2106. The time of 4pm is a hard deadline. Make sure that printouts of all code and interactions with tools are included in the hardcopy. In addition to including such printouts in the hard copy, you should also email the corresponding working code by the same deadline to skeirik2@illinois.edu.

There are four problems for you to solve. Please write your answers to each problem on separate sets of pages (using as many pages as needed for each problem). This will make it easy to grade each problem separately. All tools and code necessary for completing the exam are available from the course webpage. For questions/problems with the exam or any tools involved you can send email to (skeirik2@illinois.edu). Be sure to use the latest version of Maude (2.7.1) or else the tools may not work.

1. Define four equational theories $(\Sigma, E)$ and for each one give a $\Sigma$-equation $u = v$ such that:

   \begin{itemize}
   \item[(a)] $(\Sigma, E) \not\vdash u = v$
   \item[(b)] $\mathcal{T}_{\Sigma/E} \models u = v$.
   \end{itemize}

   For each such example you are also asked to give a precise argument explaining why (a) and (b) hold for that example. Your explanations may consist of both mathematical arguments and/or tool verifications (tool-based proofs are not required, but they can be more conclusive). For example, you might show that $\mathcal{T}_{\Sigma/E} \models u = v$ by giving a proof of it using the Maude ITP.

   Note. If you have difficulty coming up with four different theories $(\Sigma, E)$, it is OK for you to use fewer than four theories, and to give instead several different equations for some theories such that (a) and (b) holds for each of those equations, provided you still can give four examples. Extra credit will be given to students who do indeed exhibit four different theories and corresponding equations such that (a) and (b) hold for each pair.

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 $\text{fair}.l$, defined by the equality:

   $$\text{fair}.l = (\Box \Diamond \text{enabled}.l) \rightarrow (\Box \Diamond \text{taken}.l)$$

   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 $\text{enabled}$ state predicate can be defined in Lecture 18 (for all rules, but obviously this can be specialized to each rule label). Defining the $\text{taken}.l$ 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. You saw an example of this tagging in the semantics of the THREADED-IMP language of Lecture 26, 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:

   $$\text{enabled}.l \leadsto \text{taken}.l$$
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 \( enabled.l \rightarrow taken.l \) 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 single 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. Recall that in Homework 7 you were asked to specify in LTL and prove by model checking for three specific initial states \( init1, init2, \) and \( init3 \) that the COMM-CHAN example satisfied two LTL properties parametric on the list \( Q \) initially in the sender’s buffer, namely:

- the parametric invariant in-order-reception, and
- using a parametric atomic proposition called received that holds when the protocol has finished if the exact same list originally held by the sender is now in the buffer of the receiver, the parametric property expressing that such a state is indeed always reached from the given initial states.

What you could not do by explicit-state LTL model checking was to prove that these two LTL properties hold for all initial states of the general form \(< Q : 0 \mid \text{null} \mid 0 : \text{nil} >\) with \( Q \) a list, for the simple reason that there is an infinite number of such initial states.

Here is where reachability logic comes in as a more powerful, deductive method to prove parametric properties of a concurrent system that may hold for an infinite set of initial states. Given the slightly modified version of COMM-CHAN below:

```plaintext
mod COMM-CHAN is
    sorts Nat Msg MsgMSet Elt NeList List CHState .
    subsort Msg < MsgMSet .
    subsort Elt < NeList < List .

    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor] .
    ops a b c d : -> Elt [ctor] .
    op _;_ : List List -> List [ctor assoc] .
    op nil : -> List [ctor] .
    op null : -> MsgMSet [ctor] .
    op _ _ : MsgMSet MsgMSet -> MsgMSet [ctor assoc comm id: null] .

    vars N M I J : Nat . var MS : MsgMSet . vars L L' : NeList . var P Q : List .
    vars A B : Elt .

    eq Q ; nil = Q [variant] .
    eq nil = Q [variant] .

    rl [send] : < P @ A ; L : I \mid \text{null} \mid J : Q > => < P @ L ; s(I) \mid [A,I] \mid J : Q > .
    rl [send] : < P @ A : I \mid \text{null} \mid J : Q > => < P @ nil ; s(I) \mid [A,I] \mid J : Q > .
    rl [rec] : < P @ Q ; I \mid MS [A,J] \mid J : nil > => < P @ Q ; I \mid MS \mid s(J) : A > .
    rl [rec] : < P @ Q ; I \mid MS [A,J] ; L' > => < P @ Q ; I \mid MS \mid s(J) : L' ; A > .
endm
```
where the only modifications are: (i) declared constants are used instead of Qid’s as list elements; (ii) naturals are defined in the module itself; (iii) non-empty lists are used to avoid using the id: attribute for nil (because id: is not currently supported by Maude for unification modulo associativity); (iv) there are two rules for sending and two for receiving because the id: attribute for nil is not used; (v) the [send] rules are now restricted, so that they will only fire when the channel is empty; and (vi) in this version the sender keeps a copy of the original list Q that is sending to the receiver. Notice that the copy of Q kept by the sender never changes by subsequent transitions.

What you are now asked to do is to the following:

(i) Specify in reachability logic the parametric invariant in-order-reception, from any initial state $< Q @ Q : 0 | \text{null} | 0 : \text{nil} >$ specified as a reachability formula parametric on Q. Let us call the above pattern defining the initial states $S_0$.

(ii) Prove using the RL tool the formula you have specified in (i) by taking into account the following suggestions:

- Recall from Lecture 23 that proving invariants in reachability logic requires a prior theory transformation.
- Note in-order-reception is an invariant, but is too weak for a proof to be carried out. What you are asked to do is to:
  - (a) define a stronger invariant $B$ that can more easily be seen to be preserved by the rewrite rules;  
  - (b) use the Corollary in page 7 of Lecture 24 to prove that $B$ is an invariant from the set $S_0$ of initial states mentioned above, and
  - (c) give a high-level argument justifying why the set of states defined by your stronger invariant $B$ is contained in the set of states defined by the in-order-reception invariant (a detailed proof is not required).
- When defining $B$ you may find it useful to observe that in any states reachable from $S_0$ the channel is either empty of contains a single message.

(iii) Using the received property, parametric on Q, now specified as a pattern formula that holds when the protocol has finished if the list Q is now in the buffer of the receiver, express as a reachability formula parametric on Q that such a state satisfying received is indeed always reached from the (parametric) initial state $< Q @ Q : 0 | \text{null} | 0 : \text{nil} >$. Note that you can use without proof that the above send and receive rules are terminating, so the “always” qualification in the statement that “such a state is indeed always reached from the (parametric) initial state $< Q @ Q : 0 | \text{null} | 0 : \text{nil} >$” is unproblematic for this example.

(iv) Prove using the RL tool the formula you have specified in (iii) by taking into account the following suggestions:

- Proving the reachability formula in (iii) directly is problematic. Instead you should use the following generalize and conquer proof strategy:
  - To prove a reachability formula $A \rightarrow^\circ B$ prove instead: (a) $A' \rightarrow^\circ B$, where $A'$ defines a bigger set of states than $A$; and (b) that indeed $A'$ defines a bigger set of states than $A$ by using the subsumed command.
- Note that, since in this case your reachability formula is not an invariant, you must use the rewrite theory COMM-CHAN itself: here you should not use any theory transformation as done in the case of invariants.
- Take into account that the RL tool requires you to specify the set of terminating states by a pattern, which in this case should be the terminating states for the theory COMM-CHAN itself.
- Depending on how you specify the formula in (iii), you may need to use case analysis as your first proof step. In general, we must avoid letting goals unify with the terminating states before we can prove them, since that would mean we terminated without reaching our desired target. See lecture 24 pages 12-17 for examples of proofs where a case analysis was required.

**Warning. You should not add any equations either to COMM-CHAN itself, or to the transformed version of COMM-CHAN needed to prove the invariant.**
4. The Pipeline Java program below

```java
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.");
    }
}
```
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.