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 \( \text{fair}.l \), 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 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:

\[
\text{enabled}.l \rightsquigarrow \text{taken}.l
\]

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

\[
\text{enabled}.l \rightarrow \Diamond \text{taken}.l
\]

You are asked to answer the following questions about the relationships between: (a) the property \( \text{fair}.l \), (b) the \( \text{enabled}.l \rightsquigarrow \text{taken}.l \) property, and (c) the property \( \text{enabled}.l \rightarrow \Diamond \text{taken}.l \): (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. Consider the following module which defines a (somewhat problematic) version of the famous dining philosophers problem using the \texttt{THREADED-IMP} language. In the traditional formulation of the problem, there are a group of \( N + 2 \) philosophers sitting around a circle table (for \( N \) some natural number) and between each pair of philosophers there is a single chopstick. In order to eat, each philosopher must first pick up the chopstick on his left and then on his right.

In the \texttt{THREADED-IMP} version below, the chopsticks are represented by variables \( a, b, c, \) and \( d \). A variable set to 0 represents an unused chopstick. A philosopher ”picks up” a chopstick by setting the respective variable to any non-zero value.

You of course need all the modules defining \texttt{THREADED-IMP}, which are available with all other files for this homework. There are three possible initial states we will consider:

\begin{verbatim}
init-thds(0)   | init-mem | 0
init-thds(1)   | init-mem | 0
init-thds(1 + 1)| init-mem | 0
\end{verbatim}

This problem has several parts:

[1][a] Define a state proposition, \texttt{two-eat}, that holds whenever two philosophers are ”eating” at the same time, i.e. when two threads execute the ”eat” statement

[1][b] Define a parameterized state proposition, \texttt{neighbors-eat}(\texttt{N}), where \( N \) is a \texttt{Nat}, which holds whenever two neighboring philosophers are ”eating” at the same time, where the number of philosophers is \( N + 1 \)

[2][a] Using the Maude \texttt{search} command or LTL model checking, from each initial state, check whether they can reach a state that satisfies \texttt{two-eat}

[2][b] Using the Maude \texttt{search} command or LTL model checking, from each initial state, check whether they can reach a state that satisfies \texttt{neighbors-eat}(\texttt{N}) where the initial state contains \( N + 1 \) philosophers

If [2][b] has a solution state (i.e. if any initial state can reach a state where \texttt{neighbors-eat} holds), we say that two philosophers were ”impolite,” to each other, since they used the same chopstick to eat at the same time. Answer the next question only if your answer to [2][b] shows that philosophers can reach an impolite state.

[3] Describe in your own words why the philosophers can reach an impolite state.

\textbf{Note:} You may not modify any definition in the \texttt{DINPHIL} module. Your predicate definitions may not change the meaning/semantics of any IMP program. Your predicate definitions may however use the \texttt{[owise]} attribute. The template below is included for your convenience and is also available with this homework.

\begin{verbatim}
mod DINPHIL is pr THREADED-IMP-SEMANTICS .
op eat : -> TermStmt  [ctor] .
op phil : Id Id Nat  -> Thread .
op init-mem : -> Memory .
var I I' : Id . var N : Nat .

eq init-mem       = {[a,0] [b,0] [c,0] [d,0]} .
eq init-thds(0)   = phil(a,b,0) phil(b,a,1) .
eq init-thds(1)   = phil(a,b,0) phil(b,c,1) phil(c,a,2) .
eq init-thds(1 + 1)= phil(a,b,0) phil(b,c,1) phil(c,d,2) phil(d,a,3) .
eq phil(I,I',N) =
   {while true do
     while I  = 1 do skip od ;
     I  := 1 ;
     while I' = 1 do skip od ;
     I' := 1 ;
     eat ;
     I  := 0 ;
     I' := 0
\end{verbatim}
od | N} .

--- define predicates two-eat and neighbors-eat inside this module
mod DINPHIL-CHECK is inc DINPHIL . inc (MODEL-CHECKER) *
 (sort Nat to Nat*, sort NzNat to NzNat*,
  op _+_ to _.+_, op _*_ to _.*_, op _<_ to _.<__, op _<=_ to _.<=_) .
subsort ThreadedImpState < State .
--- INSERT CODE BELOW
endm

--- INSERT CODE ABOVE
endm

--- search/LTL commands which perform the verification above
--- can two people eat at the same time?
--- include (3) commands to verify this property for each initial state
--- can two neighbors eat at the same time?
--- include (3) commands to verify this property for each initial state

4. 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);  // fix: remove parens
           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).