1.

a) We intend to model a pair of traffic lights at an intersection which will change based on the demand of the emergency vehicles, and cards and pedestrians. The lights will have three colors: red, green, and yellow. They will also have a pedestrian symbol which will either be: on, flashing, or off. The pedestrian light will be on when the light goes to green, then it will flashing before the light turns yellow (but is still green), and it will turn to off when the light turns yellow or red. The lights will never be green in both directions, and a light will stay green until there is demand for the other light to go green or an emergency alarm is raised in the other direction. Demand is an abstraction of both the walk button and the weight sensors, because for the purposes of the light they are the same. When an emergency alarm is raised in one direction while the light in the opposite direction is green, then that light will immediately go to yellow and its pedestrian light will go to off, skipping the flashing state.

e)

\*\*\* This is NuSMV 2.4.3 (compiled on Fri Apr 2 20:06:42 UTC 2010)

\*\*\* For more information on NuSMV see <http://nusmv.irst.itc.it>

\*\*\* or email to <nusmv-users@irst.itc.it>.

\*\*\* Please report bugs to <nusmv@irst.itc.it>.

-- specification AG !(lightEW.color = green & lightNS.color = green) is true

-- specification AG !(lightEW.color = green & lightNS.walking = on) is true

-- specification AG (lightEW.color = green -> AF lightEW.color = red) is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

demandNS = 0

demandEW = 0

emergencyNS = 0

emergencyEW = 0

entering = 1

exiting = 0

initStatusEntering = myentering

initStatusIdle = myidle

lightEW.color = red

lightEW.walking = off

lightEW.status = myentering

lightNS.color = red

lightNS.walking = off

lightNS.status = myidle

-> Input: 1.2 <-

\_process\_selector\_ = lightEW

running = 0

lightNS.running = 0

lightEW.running = 1

-- Loop starts here

-> State: 1.2 <-

entering = 0

initStatusEntering = myidle

lightEW.color = green

lightEW.walking = on

lightEW.status = myidle

-> Input: 1.3 <-

\_process\_selector\_ = main

running = 1

lightNS.running = 0

lightEW.running = 0

-- Loop starts here

-> State: 1.3 <-

-> Input: 1.4 <-

\_process\_selector\_ = lightNS

running = 0

lightNS.running = 1

lightEW.running = 0

-- Loop starts here

-> State: 1.4 <-

-> Input: 1.5 <-

\_process\_selector\_ = lightEW

running = 0

lightNS.running = 0

lightEW.running = 1

-- Loop starts here

-> State: 1.5 <-

-> Input: 1.6 <-

\_process\_selector\_ = main

running = 1

lightNS.running = 0

lightEW.running = 0

-> State: 1.6 <-

-- specification G ((lightEW.color = red & demandEW) -> F lightEW.color = green) is true

-- specification G ((emergencyNS & Y !emergencyNS) -> F lightNS.color = green) is true

f) We are showing that our lights will not go red unless there is a demand or an emergency for the other direction by showing that there is a case that a light will stay green indefinetely as long as there is no demand for the other way and no emergency.

2.a) Yes, because b will appear on every path that starts at b and ends at the exit node.

b)No. Take the a shortest path from *b* to exit node *z*, where a is an intermediate node. Call this path A, and split it up into two parts where A1 is *b* to *a* and A2 is a to *z*. *b* cannot be an intermediate node in A2, because if it was then A2 can be split up into two different parts, A3 which is *a* to *b* on A2, and A4 which is *b* to *z* on A2. A4 is then a shorter path from *b* to *z* than A was because we removed the sub-path that included *a*. Therefore there must either be a path from a to *z* which does not include *b* or a path from *b* to *z* which does not include *a*. This means that a *tails* b and b *tails* a cannot hold true for two different nodes a and b.

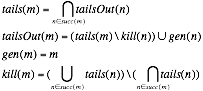
c) c *tails* a, because b *tails* a then all paths from *a* to exit node *z* must pass through b and go to z. We know from c *tails* b that all paths which go from *b* to *z* include *c*, there fore all paths which go from *a* to *z* include *c*.

d) Assuming *c* and *b* are distinct nodes, either c *tails* b or b *tails* c, but not both. Assume that neither *c* tails *b* nor *b tails c*. There exists a path from *c* to exit node *z*, that does not contain *b*, and one from *b* to *z* that does not contain *c.* There also must be a path from *a* to *b* to *z* that does not contain *c* or a path from *a* to *c* to *z* which does not include *b*, because *b* and *c* are not the same node. If there is one from *a* to *b* to *z* which does not include *c* then *c* does not tail a, and if there is a path from *a* to *c* to *z* that does not include *b* then *b* does not tail *a.* Therefore it is not true that neither *c* tails *b* and *b* tails *c* are true, therefore one of them must be true and one must be false.

e) The immediate post dominator of a node m is one which is tailed by every other node in the set tails(m), excluding m, where m is not the exit node.

f) *tails(m)* is equal to the intersection of all of the *tails(n)* sets for every successor n of node m. Any node *a* that post-dominates *m* must also post-dominate any successor of *m* between *m* and *a*. Thus *a* must post-dominate all of the successors of *m* and thus *a* would be in the intersection of the sets *tails(n)* for successors *n* of *m*.

g)



This flow analysis would be classified as *backward* and *all-path*.

h)

**Algorithm** Post-Dominance

**Input**:

A control flow graph G = (nodes, edges) with a distinguished root node start.

pred(n) = {m ∈ nodes | (m,n) ∈ edges}

succ(m) = {n ∈ nodes | (m,n) ∈ edges}

gen(n) = {m}

kill(n) = (union of tails(n) for n ∈ succ(m)) \ (intersection of tails(n) for n ∈ succ(m))

**Output**:

tails(n) = set of post-dominators of n

for n in nodes loop

tailsOut(n) = set of all nodes

end loop

workList = nodes;

while ( workList != {}) loop

n = any node in workList;

workList = workList \ {n};

oldVal = tailsOut(n);

tails(n) = intersection of tailsOut(m) for each m in succ(n);

tailsOut(n) = (Avail(N) \ kill(n)) union gen(n);

if(tailsOut(n) != oldVal) then

workList = workList union succ(n);

end if;

end loop;