CS525 Project Presentation

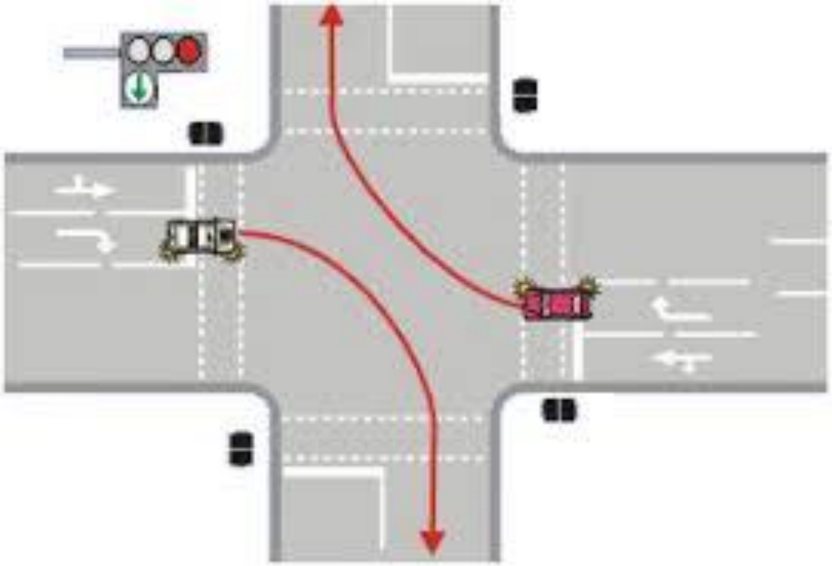
Designing and verifying models using NuSMV

Group A:

Shaurya Gomber	160101086
Rishabh Jain	160101088

Problem Description

- Junction of 2 perpendicular roads.
- Cars coming from all directions.
- Cars can take right as well.
- Design a model that handles traffic reaching this junction.



Our Approach

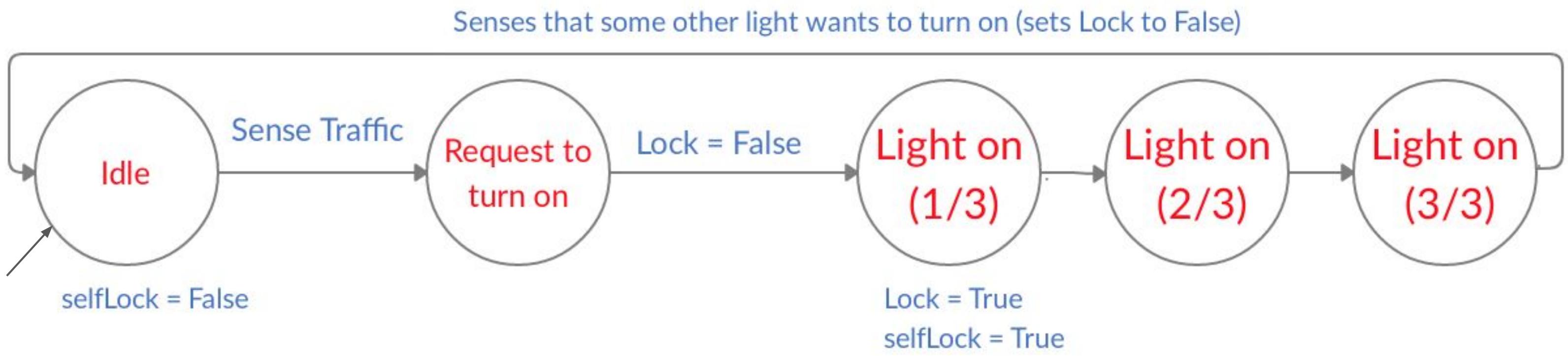
- As cars can turn also, allow cars from one direction at a time.
- We need **4** lights to achieve this.
- We will create a TrafficLight module and use 4 instances of it.
- In our design, whenever a light senses traffic, it sets requireLight true (goes to "request to turn on" state from idle state)

Ensuring Safety

- Safety condition is that no 2 lights should be on simultaneously.
- We can't turn a light on as soon as it requests as some other light may be on at that time.
- We implemented a **mutex lock** and the lights which requested to turn on had to acquire this before turning on.
- Implemented using a global variable lock which can be acquired when it is false.
- Light which acquired the lock turned itself on and set lock to True.
- Lock was set to false when the light turned off.

Model of the TrafficLight

- selfLock variable is private to each light and is true when it acquires the mutex lock.
- Once a light is on, it stays on for atleast 3 clock cycles to give time for a car to pass.
- Once a light has been on for its maximum time, it switches off only when it sees that other lights need to turn on.



Code for TrafficLight module :

```
MODULE TrafficLight(mySenseCar, myReqLight, myLight, lock, otherReq1, otherReq2, otherReq3)
VAR
    selfLock : boolean;
ASSIGN
    next(mySenseCar) := {TRUE, FALSE};

    next(myReqLight) := case
                            mySenseCar & !myReqLight & (myLight=off) : TRUE;
                            myLight!=off : FALSE;
                            TRUE : myReqLight;
                        esac;

    next(myLight) := case
                        myReqLight & selfLock & (myLight=off) : oneThird;
                        myLight=oneThird : twoThird;
                        myLight=twoThird : full;
                        (myLight=full) & (otherReq1 | otherReq2 | otherReq3) : off;
                        TRUE : myLight;
                    esac;

    next(lock) := case
                    myReqLight & !lock : TRUE;
                    selfLock & (myLight=off & !myReqLight) : FALSE;
                    TRUE : lock;
                  esac;

    init(selfLock) := FALSE;
    next(selfLock) := case
                    myReqLight & !lock : TRUE;
                    selfLock & (myLight=off & !myReqLight) : FALSE;
                    TRUE : selfLock;
                  esac;
```

Ensuring Safety

- To make sure that the model presented above is **SAFE**, the following specifications were written and tested.

```
-- Safety : No two lights are on together.
LTLSPEC G ! (northLight!=off & eastLight!=off)
LTLSPEC G ! (northLight!=off & westLight!=off)
LTLSPEC G ! (northLight!=off & southLight!=off)
LTLSPEC G ! (eastLight!=off & westLight!=off)
LTLSPEC G ! (eastLight!=off & southLight!=off)
LTLSPEC G ! (westLight!=off & southLight!=off)
```

- All these passed and thus we are sure that our system is safe.

Ensuring Liveness

- Liveness means that once a light senses traffic and requests to turn on, it should eventually turn on.

- If this is not true, then it can lead to an indefinite jam which is not desirable.

- This means, that our model should satisfy the following specifications:

```
-- Liveness
LTLSPEC G (senseFromNorth -> F(northLight!=off))
LTLSPEC G (senseFromEast -> F(eastLight!=off))
LTLSPEC G (senseFromWest -> F(westLight!=off))
LTLSPEC G (senseFromSouth -> F(southLight!=off))
```

- The model described in above slide **fails** all these tests.
- The counterexamples were analyzed to see why this happened.

Fairness Problem

- The 4 instances of TrafficLight were made using the **process** keyword (as seen below)

```
light1 : process TrafficLight(senseFromNorth,requireNorthLight,
light2 : process TrafficLight(senseFromEast,requireEastLight,ea
light3 : process TrafficLight(senseFromWest,requireWestLight,we
light4 : process TrafficLight(senseFromSouth,requireSouthLight,
```

- The verifier, at each step, non-deterministically chooses one of the lights and does one step of its automata.

- The failed cases were the once were, one light say northLight was picked and it sensed traffic and set its requireLight to true but then was never picked again.

- This happens because the verifier chooses which light to run at every time step non-deterministically.

- This can cause it to be **unfair** to a light by not picking it often.

Solution to Fairness Problem

- To sole the Fairness problem, we add **Fairness Running** to the TrafficLight module.
- This ensures that the properties are checked only on those paths where all the lights are picked (running) infinitely many times.
- After adding this, the liveness tests were run again and all of them **failed, again :(**
- The counterexamples were analyzed and a new type of problem was detected.

Starvation Problem

- The failed case was something like this :
    - Light2 requested to turn on and it did.
    - While it was on, Light1 and Light3 requested to turn on.
    - Once Light2 turned off, Light3 acquired the lock and turned on.
    - While Light3 was on, Light2 again sensed traffic and requested to turn on.
    - Once Light3 turned off, Light2 took the lock and turned on.
    - Meanwhile Light3 sensed traffic and requested to turn on and .....

- This way, traffic came coming to Light2 and Light3 and then started to juggle the lock between themselves.

- The poor Light1 (and the people in cars stuck there) could only see the light oscillating from Light2 to Light3 and were **starved** for light.

- This happened because there was no bound on how long a light must wait to get the lock once it requests for it.

- Lights requesting after Light1 also got the lock first.

Solution to Starvation Problem

- Using a **FIFO ordering** while giving the lock to the lights.
- Whenever a light requests to turn on, push it at end of the queue.
- Whenever the lock is free, give it to the light at the front of queue and pop it from the queue.
- An **array of size 4** is used to implement queue.
- PushIndex specifies end of queue while popIndex specifies the front.
- Queueing ensures that no light will starve as all lights requesting after a a certain light will get lock after that light.
- This further ensures finite waiting times for all lights trying to acquire the lock.

Modified code to ensure liveness (Contd.)

```
MODULE TrafficLight(mySenseCar, myReqLight, myLight, lock, otherReq1, otherReq2, otherReq3, queue, pushI, popI,myID)
VAR
    selfLock : boolean;
ASSIGN
    next(mySenseCar) := {TRUE, FALSE};

    next(myReqLight) := case
                            mySenseCar & !myReqLight & (myLight=off) : TRUE;
                            myLight!=off : FALSE;
                            TRUE : myReqLight;
                        esac;

    next(myLight) := case
                        myReqLight & selfLock & (myLight=off) : oneThird;
                        myLight=oneThird : twoThird;
                        myLight=twoThird : full;
                        (myLight=full) & (otherReq1 | otherReq2 | otherReq3) : off;
                        TRUE : myLight;
                    esac;

    next(queue[0]) := case
                            (pushI=0) & mySenseCar & !myReqLight & (myLight=off) : myID ;
                            TRUE : queue[0];
                        esac;
    next(queue[1]) := case
                        (pushI=1) & mySenseCar & !myReqLight & (myLight=off) : myID ;
                        TRUE : queue[1];
                    esac;
    next(queue[2]) := case
                        (pushI=2) & mySenseCar & !myReqLight & (myLight=off) : myID ;
                        TRUE : queue[2];
                    esac;
    next(queue[3]) := case
                        (pushI=3) & mySenseCar & !myReqLight & (myLight=off) : myID ;
                        TRUE : queue[3];
                    esac;
```

```
    next(pushI) := case
                        ! (mySenseCar & !myReqLight & (myLight=off)) : pushI;
                        pushI = 0 : 1;
                        pushI = 1 : 2;
                        pushI = 2 : 3;
                        pushI = 3 : 0;
                        TRUE : pushI;
                     esac;


    next(lock) := case
                     (queue[popI] = myID) & myReqLight & !lock : TRUE;
                     selfLock & (myLight=off & !myReqLight) : FALSE;
                     TRUE : lock;
                   esac;

    next(popI) := case
                     ! (selfLock & (myLight=off & !myReqLight)) : popI;
                     popI = 0 : 1;
                     popI = 1 : 2;
                     popI = 2 : 3;
                     popI = 3 : 0;
                     TRUE : popI;
                 esac;

    init(selfLock) := FALSE;
    next(selfLock) := case
                     (queue[popI] = myID) & myReqLight & !lock : TRUE;
                     selfLock & (myLight=off & !myReqLight) : FALSE;
                     TRUE : selfLock;
                   esac;

FAIRNESS running
```

Ensure Liveliness

- Solving the Fairness Problem and Starvation Problem ensures liveness.
- All the specifications mentioned below are followed by the model

```
-- Liveness
LTLSPEC G (senseFromNorth -> F(northLight!=off))
LTLSPEC G (senseFromEast -> F(eastLight!=off))
LTLSPEC G (senseFromWest -> F(westLight!=off))
LTLSPEC G (senseFromSouth -> F(southLight!=off))
```

- It is ensured that once a light senses traffic, it always turns on to let the traffic pass by turning on.

No Strict Sequencing

- We could have easily fixed an ordering of the lights and turned them on in a round robin way.

- This would satisfy :
    - Safety : As no two lights turn on together
    - Liveness : As at all times, all lights are going to turn on in the future

- It has following disadvantages :
    - All lights treated uniformly, irrespective of their traffic.
    - A light where there is no traffic also turns on and wastes time.
    - If whole traffic would be on a single light, then the 3 lights would turn on redundantly

- So, we need to make sure we are not forcing any such ordering.

No Strict Sequencing

- We specify the following and run our model :

```
-- no strict sequencing
LTLSPEC  G ((northLight!=off) -> (F(southLight!=off) -> ( ((eastLight=off)&(westLight=off)) U (southLight!=off))) )
LTLSPEC G ((northLight!=off) -> (F(eastLight!=off) -> ( ((southLight=off)&(westLight=off)) U (eastLight!=off))) )
LTLSPEC G ((northLight!=off) -> (F(westLight!=off) -> ( ((eastLight=off)&(southLight=off)) U (westLight!=off))) )
```

- All these specifications **fail** but this time, this is a positive sign :) indicating no strict sequencing.

- Specifications mentioned above are written in a way that they check that whether in all paths, a certain light always turns on after the northLight.

- The counterexamples show that any light can turn on after the northLight and which light turns on totally depends on the traffic sensed.

- Thus, our model has no strict sequencing.

Conclusions

- Designed a traffic controller incrementally.
- Checked its correctness by using the NuSMV model verifier.
- This ppt handled the case where cars were allowed to turn.
- Writing code for case when turn is not allowed is now easy as it can be done by using just 2 lights and other things can be done easily Keeping in mind, the points discussed today.
- Codes for both the cases are present and can be used for future reference.

THANK YOU