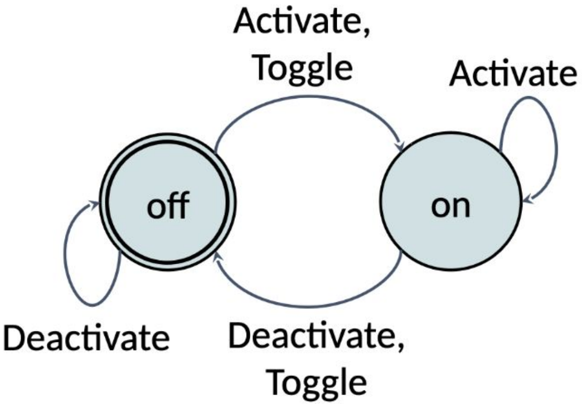
# EEcS 498 - 003 Lab 3/4

Keshav Singh

# Modelling Systems as State Machines

- From Class:
  - A state is an assignment of values to variables
  - The state space is the set of possible assignments (i.e. all states)
  - An action is a transition from one state to another
  - An execution is a sequence of states
  - A behavior is the set of all possible executions
  - Goal: prove properties about behavior of a system with state machines

# Example From Lecture




```
datatype SwitchState = On | Off
datatype Variables =
        Variables(switch:SwitchState)
predicate Init(v:Variables) {
    v.switch == Off
}
```

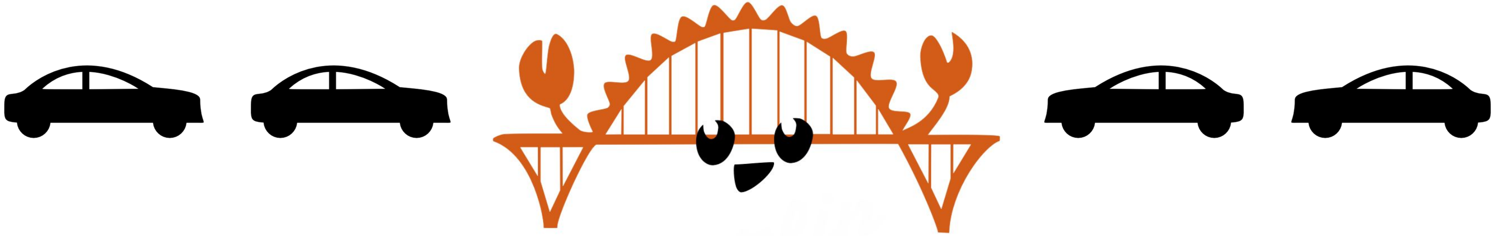
```
predicate Activate(v:Variables, v':Variables) {
    v'.switch == On
}
predicate Deactivate(v:Variables, v':Variables)
{
    v'.switch == Off
}
predicate Toggle(v:Variables, v':Variables) {
    v'.switch == if v.switch.On? then Off else
On
}
predicate Next(v:Variables, v':Variables) {
    || Activate(v, v')
    || Deactivate(v, v')
    || Toggle(v, v')
}
```

# Exercise: Rusty Bridge with PermissionManager

- Safety: The Rusty Bridge can only support 1 car at a time without collapsing
- Implement pass based system:
    - Initially no cars at either side of the bridge
    - Cars queue up at both sides of the bridge without a pass
    - System maintains state for how many current passes are given out
    - If system records no passes are given out give one to a car at the beginning of one of the queues
    - Any car with a pass is allowed to cross the bridge
    - Passes are taken when cars exit the bridge and information recorded in the system

# Rusty Bridge with PermissionManager

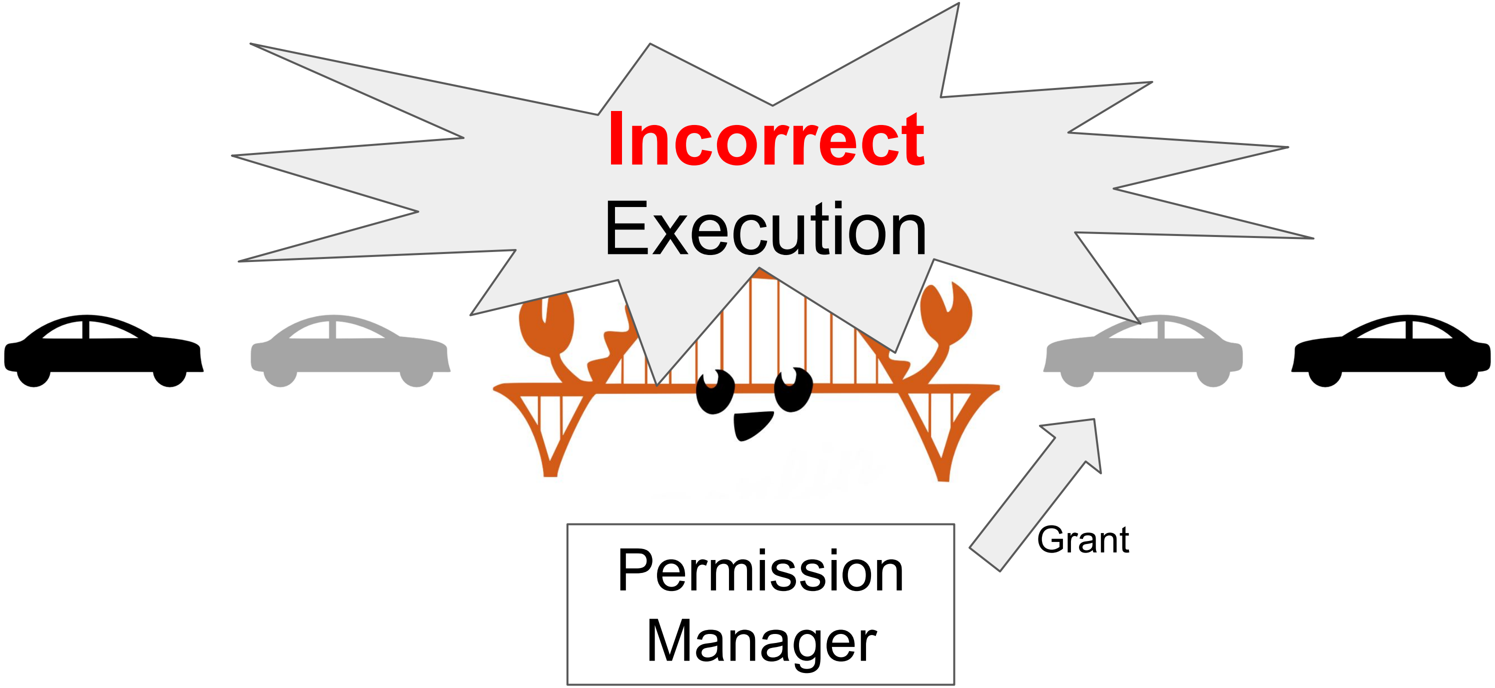
Goal: only one car on bridge at a time



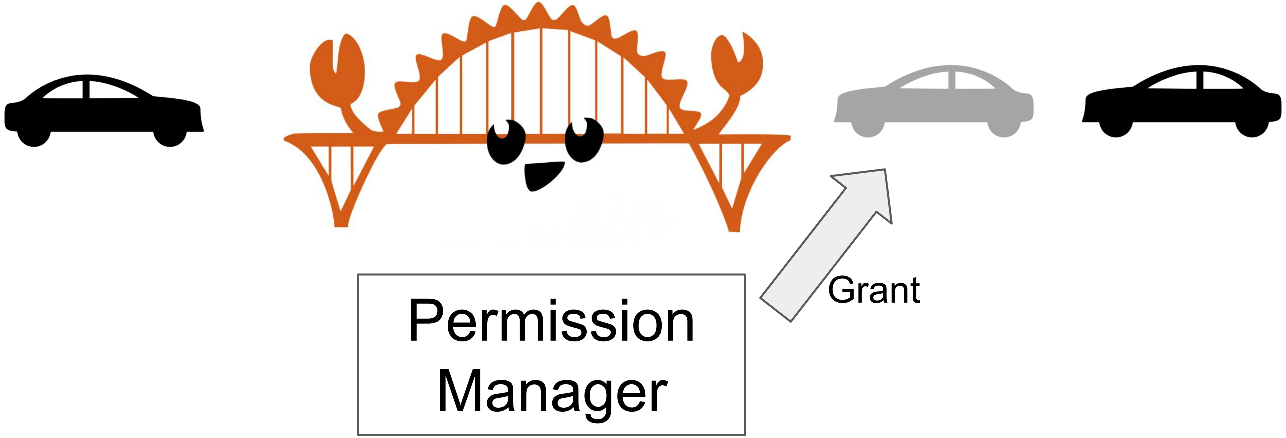
Permission Manager

# Grant Permission To Left Car

# Grant Permission To Right Car



# Incorrect Execution



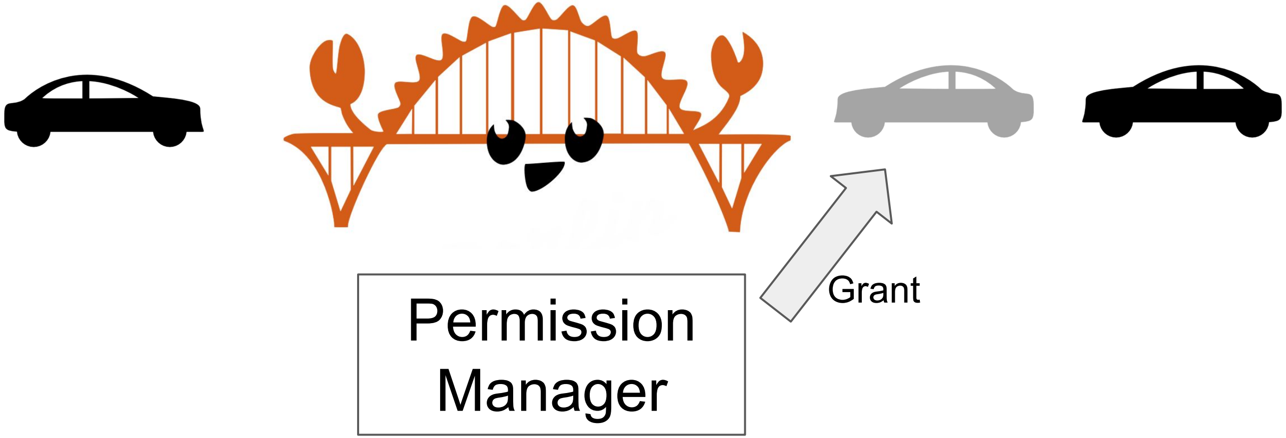
# LeftCarComingOnBridge

Permission Manager

# Car Passes the Bridge

Permission Manager

# Grant Permission To Right Car



# Implementation Details vs Modelling Details

- Implementation state is what our system actually keep (numberOfCarsWithPasses:int)
- We have other state we use for modelling/verification purposes (queue of cars, number of cars on the bridge)
- Important to not over-restrict transitions by making unnecessary assumptions in our model as this may cause our proof to be weak/ineffective
- Eg. cannot assume anything about the number of cars on the bridge when giving a pass

# General Strategy: Use an Inductive Proof

- Show that the desired property holds on state 0 & if the property holds on state k, it must hold on state k+1
- An invariant that satisfies this is called inductive
- Challenge: safety property is not always inductive
- To solve this we must further restrict our invariant so that it is inductive and implies the safety property
- This is usually an intermediate between reachable states (inductive, but hard to specify) and safe states (non-inductive, but easy to specify)
- Usually useful to add any properties about reachable states we believe make it safe

# Jay Normal Form: Helping Dafny's Automation

```
datatype Step =
  | Action1Step( <parameters> )
  | Action2Step( <parameters> )
  ...

predicate NextStep(v: Variables, v': Variables, step:Step)
{
  match step
    case Action1Step(<parameters>) => Action1(v, v', <parameters>)
    case Action2Step(<parameters>) => Action2(v, v', <parameters>)
    ...
}
predicate Next(v: Variables, v': Variables)
{
    exists step :: NextStep(v, v', step)
}
```