

---

# Finite State Machine (DFA) Based Traffic Crossing Simulator

Theory Of Computation (MC-304)

Delhi Technological University

7<sup>th</sup> May 2020

---



Anish Sachdeva  
DTU/2K16/MC/13

Anmol Vohra  
DTU/2K17/MC/18

Akshita Chander  
DTU/2K17/MC/13

## Acknowledgements

We would like to express our special thanks of gratitude to our teacher Prof Dr. Sangita kansal of the Mathematics Department who gave us a valuable opportunity to do this wonderful project on Theory of Automata and its practical application .

In this project we learnt a lot about Finite State Machines and their various real life applications including the now ubiquitous Red Lights paced at a Traffic Crossing. We learnt how red lights at a crossing are implemented in real life using transition states and timers.

We also had the opportunity to implement this theoretical concept as a simulator using Angular, TypeScript and RxJs which further introduced us to many technologies and gave us an excellent learning opportunity on how to create real-time simulators and deploy them and host them. We also had to collaborate during development which led us to use more developer centric tools.

We would also like to thank our parents and friends who helped us a lot in finalizing this project within the limited time frame.

## Undertaking

We the students of Delhi Technological University (DTU), Shahbad, Daulatpur, main Bawana Road:

1. Anish Sachdeva (DTU/2K16/MC/13)
2. Anmol Vohra (DTU/2K17/MC/18)
3. Akshita Chander (DTU/2K17/MC/13)

Hereby declare that the project created hereby and the implementation deployed here and code made available [here](#) are solely our work and no aid was taken from any extraneous persons or organizations.

1. [Code Repository](#)
2. [Traffic Crossing Simulator](#)

We also state all sources of Information and books etc. that we used as a reference in the Bibliography Section of the Project Report.

# Index

<b>Introduction</b>	<b>1</b>
<b>Motivation</b>	<b>2</b>
<b>Finite Automata</b>	<b>3</b>
<b>Deterministic Finite State Automata (DFA)</b>	<b>4</b>
<b>Moore Machine</b>	<b>5</b>
Definition	5
Relationships With Mealy Machines	5
<b>Mealy Machine</b>	<b>6</b>
Definition	6
Example	6
Applications	6
<b>Mealy vs. Moore Machine</b>	<b>8</b>
Moore Machine	8
Mealy Machine	8
<b>Moore to Mealy Machine</b>	<b>9</b>
<b>Transition Diagram For Traffic Lights</b>	<b>10</b>
State 1	11
State 2	12
State 3	12
State 4	12
<b>System Design &amp; General Specifications</b>	<b>13</b>
red-light.component.html	13
red-light.component.css	13
red-light.ts	14
red-light-color.enum.ts	14
crossing.ts	15
<b>Running the Project on Your Browser</b>	<b>21</b>
<b>Running the Project Locally</b>	<b>22</b>
<b>Examples</b>	<b>23</b>
<b>Limitations of Finite State Machine</b>	<b>24</b>
<b>Conclusion</b>	<b>25</b>
<b>Bibliography</b>	<b>26</b>



## Introduction

The project creates a simulator that mimics a crossing representing a main street and a side street. There are cars on both streets and traffic lights at the crossing that change configuration based on predefined states that have been defined as a finite State Machine.

The changing states of the traffic lights ensures that both the streets get a chance to disperse traffic.

The crossing and the traffic lights can have many different parameters whose variations can change how the cars are dispensed and which lane receives priority if at all any lane receives priority.

The project has been deployed [here](#) and allows the user to change a few of these parameters to see how the car dispersal will be effected and also see the running of the finite state machine in controlling which state the crossing is in and the variables which are required for it to move to the next state.

In a simple timer based crossing the traffic lights change color based on only the timer, but in this implementation along with a timer 2 sensors have also been added that each individually detect whether vehicles are present or not in their respective lanes or not.

The state of the crossing is then effected by the continuous timer as well as the presence of other vehicles in the lane and the following project elucidates the finite state automata and Moore machine design used to build the responsive crossing system.



## Motivation

A Traffic Crossing at an intersection changes its colors to allow cars to pass through in a systematic and orderly fashion. The purpose behind a traffic light at a crossing is to assist the depletion of traffic in an orderly and efficient manner.

But many times at a crossing the streets are not of equal weightage e.g. some street/road may be of higher priority than the other which may constitute a side street and we wish to create a system that dispenses cars from the main street at much higher rate and prioritizes this street.

In doing so we can have many variables such as:

1. Main Street Minimum Run Time ( $T_m$ ): This is the minimum amount of time for which the lights will remain green at the main street crossing.
2. Side Street Minimum Run Time ( $T_s$ ): This is the minimum time for which the side street traffic light will remain green provided there are cars in the side street. If there are no cars in the side street the traffic light will not remain green.
3. Crossing Time ( $T_c$ ): This is the amount of time taken by one car to cross the crossing when the traffic light is green for its respective position.
4. Wait Time ( $T_w$ ): This is the amount taken by the yellow light to turn green.
5. Main Street Lanes ( $L_m$ ): These are the number of lanes in the main street with minimum 1 lane and the more the lanes, the higher the rate at which cars are depleted at the crossing when the light is green. In every  $T_c$  time a car crosses in 1 lane.
6. Side Street Lanes ( $L_s$ ): These are the number of lanes in the side street and must be a minimum of 1 and the more there are lanes, the higher is the rate at which cars are depleted at the crossing when the light is green.
7. Main Street Car Arrival Time ( $A_m$ ): This is the time after which a car arrives at the main street.
8. Side Street Car Arrival Time ( $A_s$ ): This is the arrival time for the side street. A new car arrives at the side street after every  $A_s$  seconds. This can also be considered as the mean arrival time in a poisson process.

And many more variables are possible and can be added to further mimic the real world.

The motivation behind the project was to create a simulator that could take as many of these variables and give a running representation of how cars in different lanes will be cleared and at what time.

The project should also give a pictorial representation of which state the crossing currently is in and the configuration of the different traffic indicators.

## Finite Automata

A Finite Automaton is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$ , where

1.  $Q$  is a finite set called the **states**
2.  $\Sigma$  is a finite set called the **alphabet**
3.  $\delta : Q \times \Sigma \rightarrow Q$  is the **transition function**
4.  $q_0 \in Q$  is the **start state**, and
5.  $F \subseteq Q$  is the **set of accepted states**.

A finite automata can also be represented by its corresponding state transition diagram.

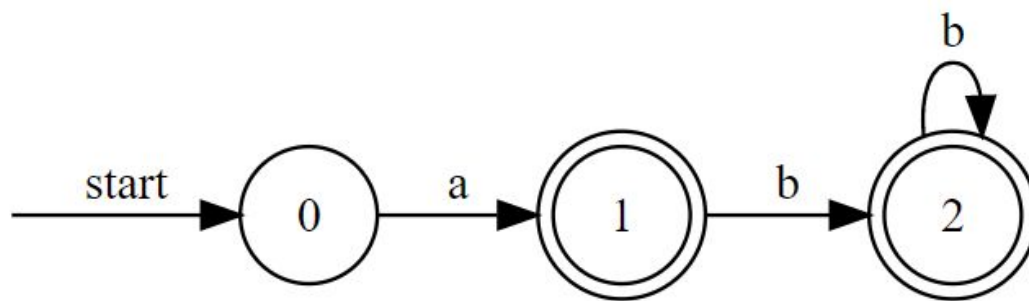


Figure 1: Finite Automata that recognizes all strings that start with exactly 1 'a' and may contain an arbitrary number of 'b' after the 'a'.

This finite state automata can also be represented with a state transition table that indicates which state it will go to after encountering a particular symbol.  $\Phi$  here denotes a null state where once the automata has entered can never come out irrespective of the symbol it encountered. It is an absorbing state.

State	a	b
$\rightarrow 0$	1	$\Phi$
1	$\Phi$	2
2	$\Phi$	2

Figure 2: The state transition table for the finite automata represented in Figure 1



## Deterministic Finite State Automata (DFA)

A deterministic finite state automata is a finite automata where given a state and an encountered symbol the machine will go into only one new (or same) state and the next state can be uniquely *determined* given the symbol and current state.

This isn't true for non-deterministic machines. Some examples of finite state automata are as follows:

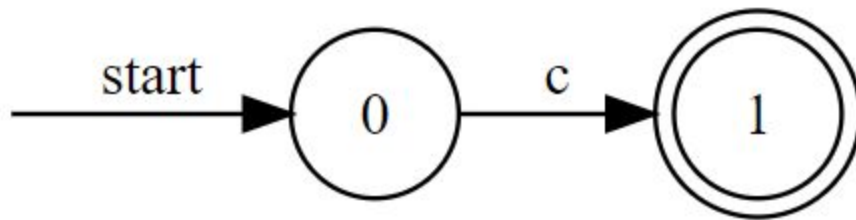


Figure 3: Deterministic Finite State Automata that recognizes only the symbol 'c'

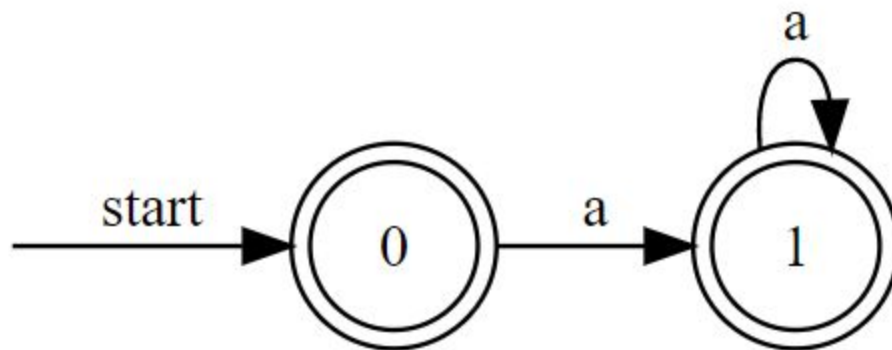


Figure 4: Deterministic Finite State Automata that recognizes the null string  $\epsilon$  or any number of 'a'

## Moore Machine

### Definition

A Moore Machine is a 6 tuple  $(S, S_0, \Sigma, \lambda, \delta, \rho)$  consisting the following:

1. A finite set of States  $S$ .
2. A start state (also called Initial State)  $S_0$  which is an element of  $S$ .
3. A finite state called the input alphabet  $\Sigma$ .
4. A finite set called the output alphabet  $\lambda$ .
5. A transition function  $\delta : S \times \Sigma \rightarrow S$  mapping pairs of a state and an input symbol to the corresponding next state.
6. An output function  $\rho : S \rightarrow \lambda$  mapping each state to the output alphabet.

A Moore Machine can be regarded as a restricted type of a Finite State Transducer.

### Relationships With Mealy Machines

As Moore and Mealy machines are both types of finite-state machines, they are equally expressive: either type can be used to parse a regular language.

The difference between Moore machines and Mealy machines is that in the latter, the output of a transition is determined by the combination of current state and current input ( $S \times \Sigma$  as the input to  $\rho$ ), as opposed to just the current state ( $S$  as the input to  $\rho$ ) when represented as a state diagram.

- For Moore Machine each node (state) is labelled with output value
- For a Mealy machine, each arc (transition) is labelled with an output value

Every Moore Machine  $M$  is equivalent to the Mealy machine with the same states and transitions and the output function  $\rho(s, \sigma) \rightarrow \sigma_M(s)$  which takes each state input pair  $(s, \sigma)$  and yields  $\rho_M(s)$  where  $\rho_M$  is  $M$ 's output function.

However, not every Mealy machine can be converted to an equivalent Moore machine. Some can be converted only to an *almost* equivalent Moore machine, with outputs shifted in time. This is due to the way that state labels are paired with transition labels to form the input/output pairs. Consider a transition  $s_i \rightarrow s_j$  from state  $s_i$  to  $s_j$ . The input causing the transition  $s_i \rightarrow s_j$  labels the edge  $(s_i, s_j)$ . The output corresponding to that input, is the label state of  $s_i$ . Notice that this is the source state of that transition. So, for each input, the outputs are already fixed before the input is received, and depends solely on the present state.

This is the original definition by E. Moore.

## Mealy Machine

### Definition

A Mealy Machine is a 6 tuple  $(S, S_0, \Sigma, \lambda, \delta, \rho)$  consisting the following:

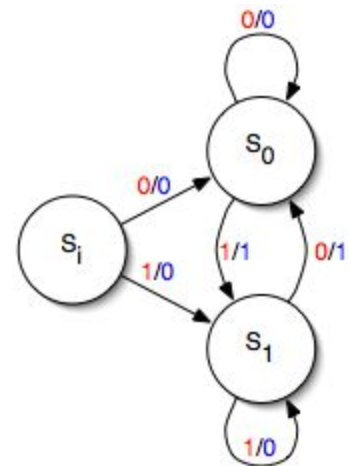
7. A finite set of States  $S$ .
8. A start state (also called Initial State)  $S_0$  which is an element of  $S$ .
9. A finite state called the input alphabet  $\Sigma$ .
10. A finite set called the output alphabet  $\lambda$ .
11. A transition function  $\delta : S \times \Sigma \rightarrow S$  mapping pairs of a state and an input symbol to the corresponding next state.
12. An output function  $\rho : S \times \Sigma \rightarrow \lambda$  mapping pairs of a state and an input symbol to the corresponding output symbol.

In some formulations the transition and the output function are coalesced into a single function  $\delta : S \times \Sigma \rightarrow S \times \lambda$ .

The state diagram for a Mealy machine associates an output value with each transition edge, in contrast to the state diagram for a Moore machine, which associates an output value with each state.

### Example

A simple Mealy machine has one input and one output. Each transition edge is labeled with the value of the input (shown in red) and the value of the output (shown in blue). The machine starts in state  $S_i$ . (In this example, the output is the exclusive-or of the two most-recent input values; thus, the machine implements an edge detector, outputting a one every time the input flips and a zero otherwise.)




### Applications

Moore/Mealy machines are DFAs that have also output at any tick of the clock. Modern CPUs, computers, cell phones, digital clocks and basic electronic devices/machines have some kind of finite state machine to control it.

Simple software systems, particularly ones that can be represented using regular expressions, can be modeled as Finite State Machines. There are many such simple systems, such as vending machines or basic electronics.

By finding the intersection of two Finite state machines, one can design in a very simple manner concurrent systems that exchange messages for instance. For example, a traffic



light is a system that consists of multiple subsystems, such as the different traffic lights, that work concurrently.

Some examples of applications:

- Number classification
- Watch with timer
- Vending machine
- Traffic light
- Bar code scanner
- Gas pumps

## Mealy vs. Moore Machine

### Moore Machine

1. In a Moore machine the output depends solely on the present state.
2. If the input character  $\sigma$  leading to the present state changes, the output symbol doesn't change.
3. Defining an equivalent Moore machine requires a higher number of states than an equivalent Mealy machine.
4. Creating an equivalent Moore circuit on the hardware requires more hardware as a higher number of states are required.
5. On input, the Moore machine changes state and after moving to the next state the new output characters are emitted. This means they respond slowly to input changes and change in emitted symbols is detected after 1 CPU clock.
6. In a Moore machine implemented on hardware the new state is emitted only once the transition has taken place and the machine is at the new state. This makes Moore machines ideal for synchronous tasks and serialized or linear-time tasks.
7. The output of Moore machines depends on the state they transition to rather than the path they took. This makes Moore machines stochastically memoryless.
8. They're relatively simpler to design.

### Mealy Machine

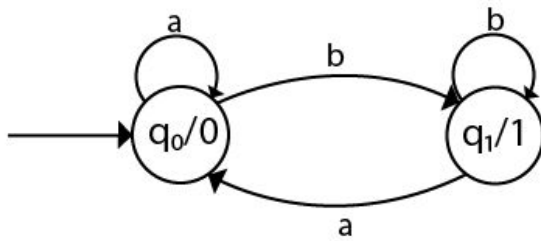
1. In a Mealy Machine the output depends both the present state and the input character  $\sigma$  leading to the present state.
2. If the input character  $\sigma$  changes, then the output symbol may change as well.
3. An equivalent Mealy machine requires less states than a Moore machine.
4. Implementing a Mealy machine on the hardware such as FPGA's and VLSI takes less hardware cores as the number of states are lower.
5. Mealy machines implemented on the hardware react much faster to input change as the new output symbol is emitted based on the new input symbol and current state and the CPU doesn't have to wait to transition to the next state to receive the new characters.
6. Mealy machines emit the new output symbol after the new input symbol is encountered even before they transition to the new state. This means that they can give a compute output even before their own operation has been completed. This makes them ideal for asynchronous applications.
7. The output of Mealy machines depends on the transition and the path taken by the machine to go from one state to the next.

## Moore to Mealy Machine

Let  $M = (Q, \Sigma, \delta, \lambda, q_0)$  be a Moore machine. The equivalent Mealy machine can be represented by  $M' = (Q, \Sigma, \delta', \lambda', q_0)$ . The output function  $\lambda'$  can be obtained by

$$\lambda'(q, \sigma) = \lambda(\delta(q, \sigma))$$

An example is as follows:



$$1. \lambda'(q_0, a) = \lambda(\delta(q_0, a))$$

$$2. \quad = \lambda(q_0)$$

$$3. \quad = 0$$

4.

$$5. \lambda'(q_0, b) = \lambda(\delta(q_0, b))$$

$$6. \quad = \lambda(q_1)$$

$$7. \quad = 1$$

$$8. \lambda'(q_1, a) = \lambda(\delta(q_1, a))$$

$$9. \quad = \lambda(q_0)$$

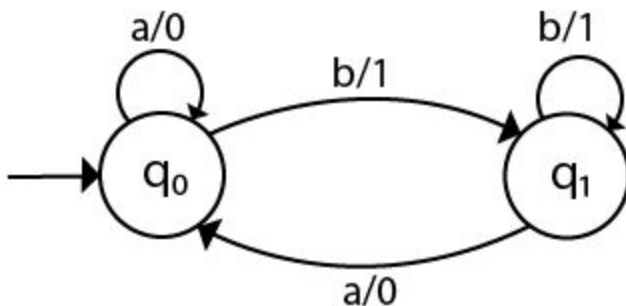
$$10. \quad = 0$$

11.

$$12. \lambda'(q_1, b) = \lambda(\delta(q_1, b))$$

$$13. \quad = \lambda(q_1)$$

$$14. \quad = 1$$

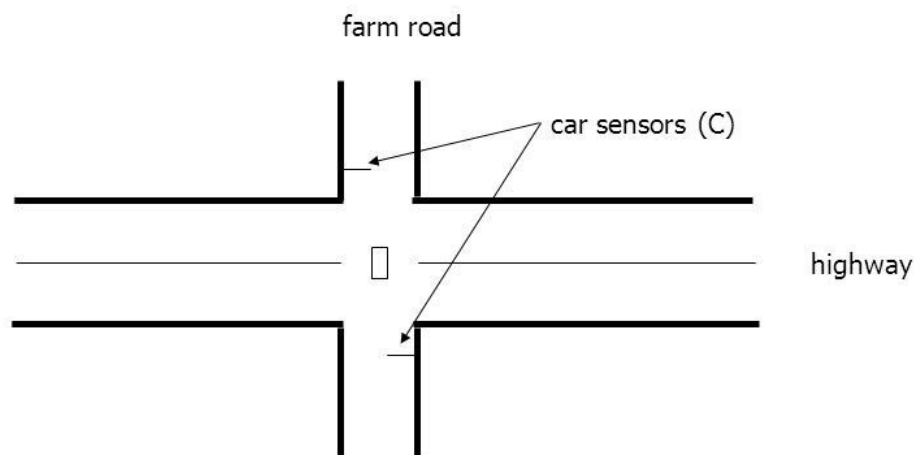


## Transition Diagram For Traffic Lights

We are assuming that there is a crossing with 4 streets running perpendicular to each other in North-South and East-West Direction. We assume that there are 2 Traffic Lights at this crossing and that at any given point of time only one light can be green and traffic will be dispensed at only one street at a time.

### [ Example: Traffic light controller ]

#### ■ Highway/farm road intersection



3

We assume that there are no underpasses or flyovers. The system can be described with the following variables. We also assume that one of the streets in this crossing is a **main street** with a higher priority than the other street - which has been relegated as the **side street**.

**$\text{cars\_in\_main\_street}(C_M)$** : This is the number of cars in the main street at any point of time and the goal of this system is to dispense the cars in this street with a priority.

**$\text{cars\_in\_side\_street}(C_S)$** : This is the number of cars in the side street.

**$\text{main\_street\_run\_time}(T_M)$** : This is the minimum duration for which the traffic light will remain green for the main street. It is possible that the traffic light remains green for a

longer period of time if there are no cars in the side street. (In that case the traffic light will not turn green for the side street and remain in the main street indefinitely).

**side\_street\_run\_time ( $T_s$ ):** This is the maximum amount of time the traffic light will remain green for the side street. It is possible that the traffic signal stays Green for a shorter period of time if the cars in the side street are dispensed at an earlier point of time.

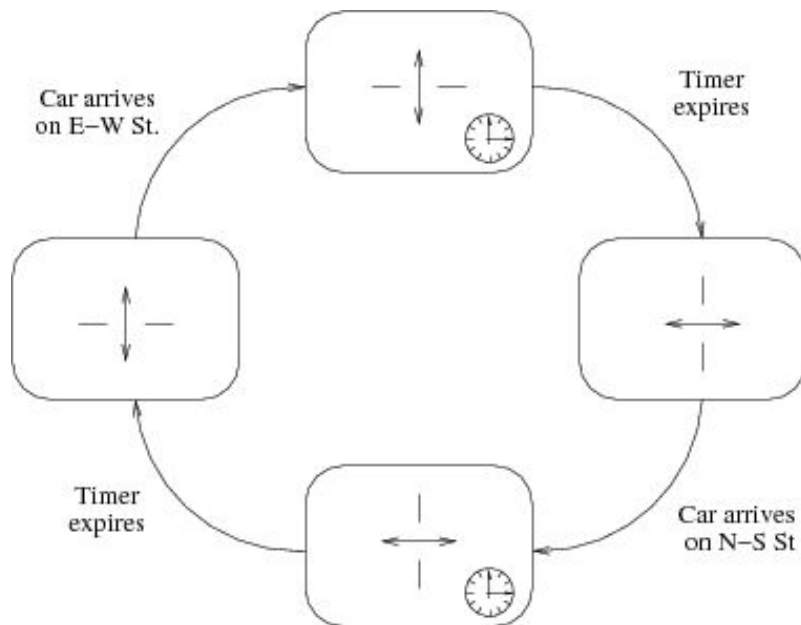
**Crossing\_time ( $T_c$ ):** This is the amount of time 1 car takes to cross this “crossing” if the traffic light for its respective lane is green. The number of cars that cross when the traffic signal is green for their lane will be  $\text{lanes\_in\_that\_street} / \text{crossing\_time}$  per second or lanes\_in\_street no. of cars every crossing\_time unit.

In this project it is assumed that the no. of lanes in both the main street and side street are 1.

**running\_speed:** This is the running speed of the simulation application. It is set to 1 by default but can be changed to 2x, 5x, 10x, 15x and 20x.

**wait\_time ( $T_w$ ):** Waiting Time is the time a traffic signal will remain in the state: Yellow before changing to the go state (State:Green).

There are 2 sensors attached to the streets - one at each street. They both sense whether there is any vehicle in their streets or not and return a Predicate value; `true` or `false` depending on whether or not there are vehicles in their respective lanes. If there are vehicles in the main street the predicate  $V_M$  is returned otherwise  $\bar{V}_M$ . Similarly if there are vehicles in the side street the predicate  $V_S$  is returned otherwise  $\bar{V}_S$ .



Now, the states are defined as follows:



## State 1

In this state the Traffic Light for the main street is at Green and the traffic signal for the side street is in Red. The system will remain in this state until the timer for main street ( $T_M$ ) has ended and there are vehicle in the side street  $V_S$ .

Once the condition  $\bar{T}_M$  and  $V_M$  is true, the system changes state and moves to State 2.

## State 2

In this state the state halt (Color:Yellow) is emitted for the Main Street Traffic Signal and the state stop (Color:Red) is emitted for the side street traffic signal. The system remains in this state for  $T_W$  amount of time in which no cars cross the street and then moves to the next state; State 3.

## State 3

In this state the automata emits the state go (Color:Green) for the side street and stop (Color:red) for the main street. The cars start dispensing from the side street. The system remains in this state for a maximum  $T_S$  time until there are cars in the side street. So as soon as the condition  $\bar{T}_S$  OR  $\bar{V}_S$  becomes true the system changes state to State 4.

## State 4

State 4 resembles state 2 very closely as this is a waiting/transition state as well where no cars are moving and a particular lane is simply in the halt (Color: Yellow) stage waiting for the signal to turn green.

In this stage the side street emits the halt (Color:Yellow) and the main street emits the stop state (Color:Red). The system remains in this state for wait time  $T_W$  after which it transitions to state 1 again.

## System Design & General Specifications

The simulation application has been Built on Google's Angular Front End Framework and written completely in TypeScript (A Typed superset of JavaScript).

The application has 2 main components. The Dashboard component and the Red Light Component. The Red Light component is just a simple component that displays a single color; Red, Green or Yellow based on the state of the Red light which is defined under the RedLightColor enumerator.

The main class is the Crossing class which contains all properties of a crossing including the 2 red light objects. It also provides an API so that external users can simulate this crossing by adjusting and modifying the variables.

### red-light.component.html

```
<div id="component-container">
  <div [ngClass]="{'red' : redLight.state === 'red'}"></div>
  <div [ngClass]="{'yellow' : redLight.state === 'yellow'}"></div>
  <div [ngClass]="{'green' : redLight.state === 'green'}"></div>
</div>
```

### red-light.component.css

```
#component-container {
  border: 1px solid grey;
  border-radius: 5px;
  padding: 5px;
}
#component-container > div {
  height: 80px;
  width: 80px;
  border-radius: 50%;
  margin-bottom: 5px;
}
.red {
  background-color: red;
}
```

```
.green {  
  background-color: green;  
}  
  
.yellow {  
  background-color: yellow;  
}
```

## red-light.ts

This is the model class that actually describes the red light and what state it's in.

```
import {RedLightColor} from './red-light-color.enum';  
export class RedLight {  
  state: RedLightColor;  
  constructor(state?: RedLightColor) {  
    this.state = state ? state : RedLightColor.GREEN;  
  }  
}
```

## red-light-color.enum.ts

This is the enumerator that defines the different colors that can be displayed on the red light.

```
export enum RedLightColor {  
  GREEN = 'green',  
  RED = 'red',  
  YELLOW = 'yellow'  
}
```

Then we have the dashboard component which is a placeholder for all the controls and the 2 red lights that have been implemented. It is also the container for the visual representations of the 4 states that this system goes through during functioning.

The main class that implements the Finite State automata and provides the simulation api is the Crossing class.

## crossing.ts

```
import {RedLight} from './red-light';
import {RedLightColor} from './red-light-color.enum';
import {interval, Observable, Subscription} from 'rxjs';
import {CrossingState} from './crossing-state.enum';

export class Crossing {
  mainStreetRedLight = new RedLight();
  sideStreetRedLight = new RedLight(RedLightColor.RED);
  carsInMainStreet = 100;
  carsInSideStreet = 50;
  mainStreetRunTime = 10;
  sideStreetRunTime = 10;
  crossingTime = 4;
  state: CrossingState;
  runningSpeed = 1;
  stateHandler$: Subscription;
  simulationRunning = false;
  stateElapsedTime: number[] = [];
  totalElapsedTimeSeconds = 0;
  crossingTimeSeconds = 0;
  waitTime = 5;

  constructor() {
  }

  stopSimulation() {
    this.simulationRunning = false;
    this.endStateHandler();
    this.state = undefined;
    this.setElapsedTimesToZero();
  }
}
```

```
}
```

```
setElapsedTimesToZero(): void {  
    this.totalElapsedTimeSeconds = 0;  
    this.crossingTimeSeconds = 0;  
}
```

```
initializeRedLights(): void {  
    this.mainStreetRedLight = new RedLight();  
    this.sideStreetRedLight = new RedLight(RedLightColor.RED);  
}
```

```
endStateHandler() {  
    if (this.stateHandler$) {  
        this.stateHandler$.unsubscribe();  
    }  
}
```

```
startSimulation() {  
    this.simulationRunning = true;  
    this.initializeRedLights();  
    this.state1();  
}
```

```
state1() {  
    this.handleTransition(CrossingState.STATE_1);  
    this.stateHandler$ = this.createTimer().subscribe((tick) => {  
        this.stateElapsedTime[0] = tick + 1;  
        this.crossingTimeSeconds = this.stateElapsedTime[0] % this.crossingTime;  
        this.totalElapsedTimeSeconds++;  
        if ((tick + 1) % this.crossingTime === 0) {  
            this.carsInMainStreet = this.carsInMainStreet === 0 ? 0 : this.carsInMainStreet - 1;  
        }  
    });  
}
```

```

    }

    if ((tick + 1 >= this.mainStreetRunTime) && (this.carsInSideStreet > 0)) {
        this.state2();
    }
});
}

state2() {
    this.handleTransition(CrossingState.STATE_2);
    this.stateHandler$ = this.createTimer().subscribe((tick) => {
        this.stateElapsedTime[1] = tick + 1;
        this.totalElapsedTimeSeconds++;
        if (tick + 1 >= this.waitTime) {
            this.state3();
        }
    });
}

state3() {
    this.handleTransition(CrossingState.STATE_3);
    this.stateHandler$ = this.createTimer().subscribe((tick) => {
        this.stateElapsedTime[2] = tick + 1;
        this.crossingTimeSeconds = this.stateElapsedTime[2] % this.crossingTime;
        this.totalElapsedTimeSeconds++;
        if ((tick + 1) % this.crossingTime === 0) {
            this.carsInSideStreet = this.carsInSideStreet === 0 ? 0 : this.carsInSideStreet - 1;
        }
        if (tick + 1 >= this.sideStreetRunTime || this.carsInSideStreet === 0) {
            this.state4();
        }
    });
}
}

```

```
state4() {  
    this.handleTransition(CrossingState.STATE_4);  
    this.stateHandler$ = this.createTimer().subscribe((tick) => {  
        this.stateElapsedTime[3] = tick + 1;  
        this.totalElapsedTimeSeconds++;  
        if (tick + 1 >= this.waitTime) {  
            this.state1();  
        }  
    });  
}
```

```
createTimer(seconds?: number): Observable<number> {  
    seconds = seconds === undefined || null ? 1 : seconds;  
    return interval(1000 * seconds / this.runningSpeed);  
}
```

```
handleTransition(state: CrossingState) {  
    this.state = state;  
    this.crossingTimeSeconds = 0;  
    if (this.stateHandler$) {  
        this.stateHandler$.unsubscribe();  
    }  
    this.setRedLightColorForMainStreet();  
    this.setRedLightColorForSideStreet();  
}
```

```
setRedLightColorForMainStreet() {  
    switch (this.state) {  
        case CrossingState.STATE_1:  
            this.mainStreetRedLight.state = RedLightColor.GREEN;  
            break;
```

```

    case CrossingState.STATE_2:
        this.mainStreetRedLight.state = RedLightColor.YELLOW;
        break;
    case CrossingState.STATE_3:
    case CrossingState.STATE_4: this.mainStreetRedLight.state = RedLightColor.RED;
    }
}

setRedLightColorForSideStreet() {
    switch (this.state) {
        case CrossingState.STATE_1:
        case CrossingState.STATE_2:
            this.sideStreetRedLight.state = RedLightColor.RED;
            break;
        case CrossingState.STATE_3:
            this.sideStreetRedLight.state = RedLightColor.GREEN;
            break;
        case CrossingState.STATE_4:
            this.sideStreetRedLight.state = RedLightColor.YELLOW;
        }
    }

    addCarInMainStreet(): void {
        this.carsInMainStreet++;
    }

    addCarInSideStreet(): void {
        this.carsInSideStreet++;
    }
}

```

The main functions made public by this class are `startSimulation()` and `stopSimulation()`. When the user calls `startSimulation()` it starts from `state_1` in





the automata and moves on to next states on the basis of the transitions that have been defined.

It maintains a global timer that ticks after every second (or even faster depending on the simulation speed defined by the user) and also two variables that record the number of cars in the side street and the main street; `carsInSideStreet` and `carsInMainStreet`.

Based on the global timer and the value of these variables it makes state transitions and as it makes these state transitions an auxiliary function gets emitted which modifies the state of the 2 Red Lights at the crossing. When the states of the red lights are modified they change color.

Another function that this timer takes care of is the dispensation of cars from lanes and as the timer is ticking and the value equals the `crossingTime` one car from that particular lane is cleared.

## Running the Project on Your Browser

The project has been deployed on Google Firebase and can be viewed [here](#).

The project code has been pushed on an online repository on Github and project code can be viewed and checked out [here](#).

The project has been published with an open source MIT license so that other people can freely view my work, extend it and also use it for their own projects.

Pull requests and suggestions can be submitted on the repository page [here](#).

## Running the Project Locally

To run this project locally you must have the following software/packages installed on your local machine:

1. [Git](#)
2. [Node](#)
3. [Angular](#)
4. [TypeScript](#)

Angular and TypeScript can be installed after installing node on your machine. To check if node and npm have been successfully installed on your machine run the following commands:

```
npm --version  
node --version
```

To add angular run the following command after adding node:

```
npm i -g @angular/angular-cli
```

To see if angular has been successfully installed run:

```
ng --version
```

To add typescript run the following command after adding node:

```
npm i -g typescript
```

Clone the repository on your machine using:

```
git clone https://www.github.com/anishLearnsToCode/red-light-automata-simulator.git
```

To run locally:

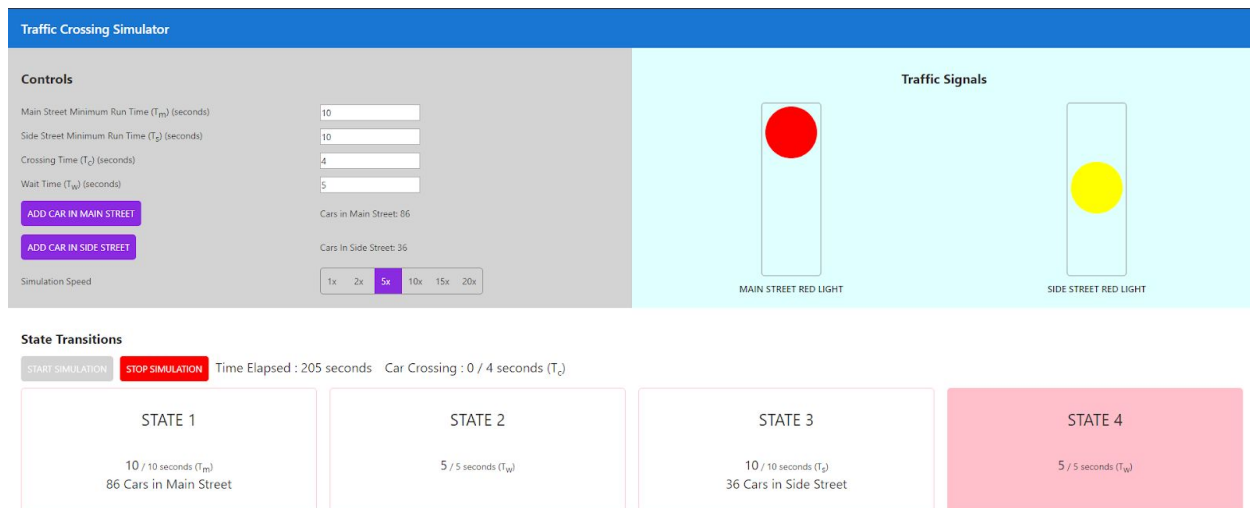
```
cd state-diagram-generator  
ng serve
```

This will now start running the web application on your localhost port 4200, which can be accessed by your web browser (prefer Google Chrome or Mozilla) at [localhost:4200/](http://localhost:4200/).

## Example

Let us take the most basic example with equal Main street run time  $T_M$  and side street run time  $T_S$  and 100 cars in the main street  $C_M$  and 50 cars in the side street  $C_S$ . We run this simulation with 5x the speed to observe that system with this configuration is cleared in ~948 seconds.

Below is an image of the running simulation.



Now, we can change the parameters to make the main street run time longer, say  $T_M = 45$  seconds. And then run the simulation with all other parameters as the same, we observe that the system is cleared in ~1200 seconds.

This seems counterintuitive as giving the high priority street more time seems like a good idea as that will dispense cars with high priority at a faster rate. Although it does do that. It dispensed the entire main street in under ~400 seconds. Due to the side street having a maximum fixed timer. It always stops and returns to the main street which causes the cars on the side street to wait longer even when there is no car in the main street.

## Limitations of Finite State Machine

The primary advantage of a Finite State Machine is its simplicity and ease of implementation for a limited number of states. Unfortunately these primary advantages are also its disadvantages.

1. A Finite State machine is memoryless, i.e. it does not know how it achieved a state or how many states it has traversed to achieve its current state. All it knows is its current state and the next state it will go in given a certain input symbol (the next state is known deterministically only in a Finite State Machine).

If we wish to build applications where the state in which the application is in is important as well as the knowledge of past states such as video games or any other knowledge based simulation a Finite State Machine cannot be used.

2. Real world scenarios can require modelling with a high number of states whose implementation may not always be feasible. In the real world the actual number of states a particle may exist in may be quite high and representing that and designing a system based on Finite State Automata with a fairly large number of states is a very cumbersome task.
3. The number of input characters that may be present in the real world or required for an application may be very high and designing a system with a high degree of input alphabets  $\Sigma$  becomes very unfeasible. Eg. in most video games more than 200+ characters and inputs are accepted from a wide variety of input devices such as the keyboard and mouse and implementing 200 transitions for every state whilst designing a video game isn't feasible.

## Conclusion

We conclude by stating that Finite State Machines can be used to design and implement systems with a finite and limited number of states, a limited character set and memory less application requirement where how the application/system responds depends on only the state it is in and the incoming character value.

A traffic light at a crossing is a perfect example of a system that is real world and whose functioning is necessary for the dispensation of traffic and the management of entire cities. A traffic light crossing can have many different variables and if the system designers wish to give some street a higher priority or wish to design a system which is equipped with both sensors and a timer controlling the system - it can be done using a Finite State Automata and using the simulator application we created we can also see how different timers, variable values and distribution of cars in side street and the main street will affect dispersion of traffic.

By classifying real world applications and problems into states and defining key interactions that can change these states we can model real world problems as Finite State Machines - encompassing Moore and Mealy machines (even Non-deterministic Automata). By modelling real world problems in this form - we can use simulations to see how that problem may very well be solved and also be able to design sustainable systems that solves the problem entirely.

Another Advantage of designing applications for systems based on Finite State automata is that if in the future the transitions were to change or the criteria for transitions was to change then instead of implementing the logic of the application all the developers would need to do is modify the transition function without changing the application at all and that would automatically update the pre-existing deployed solution without any major modifications.

So, the implementation of systems with Finite State Automata also gives developers an easy way to implement , deploy and modify complex problems.

## Bibliography

1. [Git](#) [git-scm]
2. [GitHub](#) [github-microsoft]
3. [Angular](#) [angular.io-google]
4. [Node](#) [node-org]
5. [Npm](#) [npm-org]
6. [Red-light-automata-simulator](#) [GitHub - Repository]
7. [Introduction to Theory of Computing](#) [MIT Press]
8. [Angular Hosting on Github Pages](#) [Telerik]
9. [gh-pages](#) [npm]
10. [Introduction to Theory of Computation](#) by Michael Sipser [Cengage Learning]
11. [Deterministic Finite State Automaton](#) [Wikipedia]
12. [Finite State Machine](#) [Wikipedia]
13. [Non-Deterministic Finite State Automata \(NFA\)](#) [Wikipedia]
14. [Non-Deterministic Finite State Automata](#) [Tutorials-point]
15. [Clean Code](#) by Robert C. Martin (Uncle Bob) [Amazon]
16. [GraphViz](#)
17. [DotScript](#) [Wikipedia]
18. [Mealy Machines](#) [Wikipedia]
19. [Moore Machine](#) [Wikipedia]
20. [Moore and Mealy Machines](#) [Tutorials-point]
21. [Moore Machine - An Overview](#) [Science Direct - Elsevier]
22. [Firebase Hosting Documentation](#) [Firebase - Google]
23. [Firebase CLI](#) [Firebase - Google]
24. [Difference between Mealy and Moore Machines](#) [geeks-for-geeks]
25. [Mealy vs Moore Machine](#) [vlsi-facts]
26. [Traffic Light Controller Design](#) [slideshare-linkedIn]
27. [Traffic Light Control Example](#) [university-of-pittsburgh ~kmram]
28. [Finite State Transducer](#) [Wikipedia]