ESE Homework: Paper Agartha

Tural Hasanov
Fachhochschule Dortmund
Dortmund, Germany
tural.hasanov001@stud.fh-dortmund.de

Contribution: 33%

Rahim Hashimov
Fachhochschule Dortmund
Dortmund Germany
rahim.hashimov001@stud.fh-dortmund.de

Contribution: 33%

Seda Sensoy
Fachhochschule Dortmund
Dortmund Germany
seda.sensoy003@stud.fh-dortmund.de

Contribution: 33%

I. Introduction

This paper presents an intersection management technique for self-driving vehicles in the vehicle-to-infrastructure scenario. The intersection is vital to the traffic network and is one of the leading causes of traffic accidents. Crashes at junctions may also cause major traffic backups on several routes, wasting time and money for vehicles and causing excessive air pollution. It is also estimated that almost 96 percent of intersection-related crashes had major factors ascribed to drivers, such as poor surveillance, incorrect assumption of other drivers' actions, and turning with an obscured view. Autonomous cars have recently received a lot of attention as a viable solution to traffic accidents. With the rapid advancement of autonomous cars, new IM tactics that take autonomous vehicles into consideration should be developed. To assess the traffic status, all cars are expected to be completely autonomous and capable of communicating with the intersection management unit. A static conflict matrix that reflects the possible conflict between lanes of various directions and a dynamic information list that captures the real-time occupation of each lane in the intersection determine the priority of passing through the intersection. In comparison to previous systems in the field, the intersection management unit in our approach is more like a database than a computational center, using less computational resource and more likely meeting the real-time need in high traffic circumstances.

II. Description

Intersections come in a variety of forms. For the purpose of simplicity, we will concentrate on the well-known four-way junction (or crossroad) seen in Fig 1. Our technique should be easily adaptable to a wide range of junction forms. There is a total of 12 incoming lanes. Each lane has its own id r_i (I = 0, 1, …, 11), which is numbered clockwise from the top. This intersection is straightforward since each lane has just one outgoing way. That is, the matching lane may uniquely reflect the vehicle's journey in the junction region, making the analysis considerably easier. Before we can build an Intersection Management strategy, we must first understand the traffic scenario. We suppose that:

1. Because all overtaking and lane shifting has already occurred, cars travelling on a lane must follow the lane's direction.
2. All cars are totally autonomous and capable of driving in a certain lane at a specific pace. Other modes of transportation, such as pedestrians and emergency vehicles, are not taken into account.
3. Once in communication range, all vehicles can communicate effectively with the IM unit.
4. All of the speed and time estimates made by autonomous vehicles are precise enough. These estimates serve as the foundation for later judgments made by autonomous vehicles.

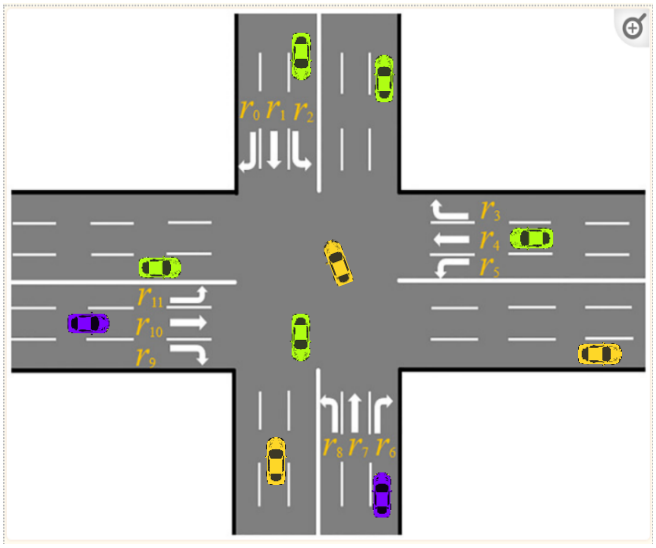

Fig. 1. Visualization

III. Modelling

The advancement of autonomous cars, particularly the inclusion of IM systems, allows for a full rethinking of the IM architecture by including additional objectives throughout the design process. Indeed, one of the most difficult aspects of developing an IM system is the lack of consensus on what a good system should be. In this part, we provide and discuss modeling of the system that we feel to be efficient. The modelling part of our project consists of several parts:

- Requirements
- Context (with Internal Blocks)
- Constraints
- State Machines

A. Requirements

We built the requirements with independent requirements and certain hierarchies. Generally, the requirements are specified for vehicles and intersection management system having in total, 11 requirements, with 5 vehicle-related and 6 system-related requirements.

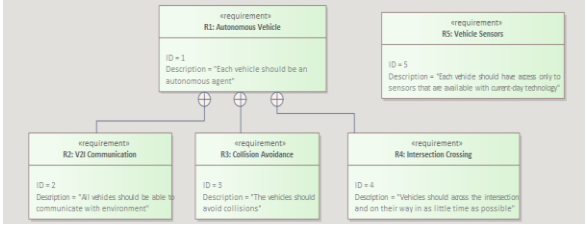

Fig. 2. Vehicle requirements

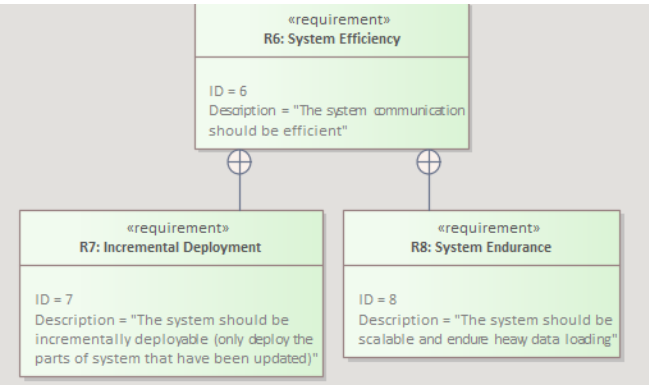

Fig. 3. System Efficiency Requirements

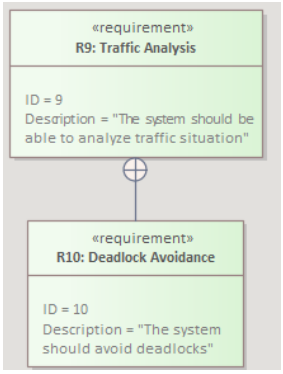

Fig 4. Analysis Requirements

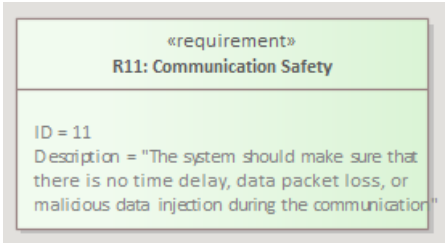

Fig. 5. Communication Safety

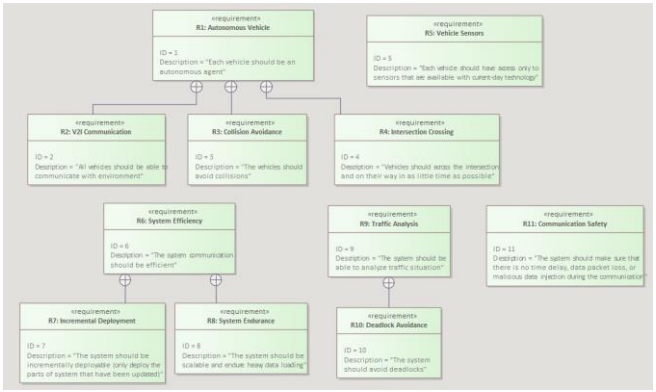

Fig. 6. Overall Requirements

B. *Architecture*

The architecture (*Fig. 7.*) has the central block Networked Traffic Control for Autonomous which connects all parts of our context together. The main blocks are Intersection Management for Autonomous Cars and Traffic Accident Detection and Reporting. the road. To have thorough insight into the architecture, we will discuss the internal parts of the blocks existing in the context.

Starting with Intersection Management for Autonomous Cars (*Fig 8.*), it uses actual Vehicle and Intersection Manager, and it has the property of Vehicle to Infrastructure Communication which enables Vehicle to communicate with Intersection Manager. Vehicle block (*Fig. 9.*) consists of the properties that are directly related to vehicle itself such as length, width, velocity, ID etc. Intersection Manager (Fig. 8.) is the most important block where most processes are implemented. It contains Intersection Control Policy which is basically our algorithm of intersection passing. Vehicle Interface is the data structure that represents vehicle intersection which holds necessary properties of vehicle for intersection passage such as ID, route, pass time, velocity etc. The Information List is used as a database to store vehicles (Vehicle Interface instances) which are in the intersection area and assigned relevant pass time, velocity etc. values as well as compare incoming vehicles with stored vehicles to calculate necessary values for new vehicles also. Conflict Matrix is a tool that helps our calculations of vehicle properties by route conflict. Two routes are said to be in conflict if they intersect at some point within the traffic intersection area. Vehicle routes can be represented by lanes. As a result, we can say whether two lanes are in conflict or not. We can determine whether two lanes conflict with each other and then we can create a conflict matrix for all lanes. We will discuss conflict matrix in detailed in the following chapters. Accident Sensing Manager (Fig. 10.) is constantly getting input images and pass those images to Image Processing unit to get results whether there is an accident at the intersection or not, and due to that the connection between Input Images and Image Processing is analogous.

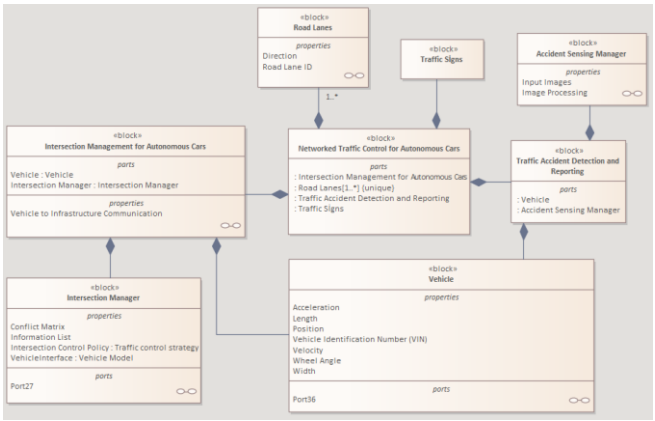

Fig. 7. Context Diagram

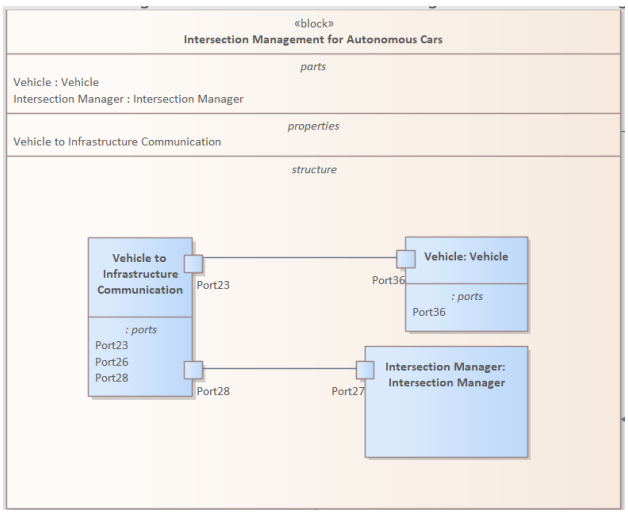

Fig. 8. Intersection Management for Autonomous Cars Internal Block Diagram

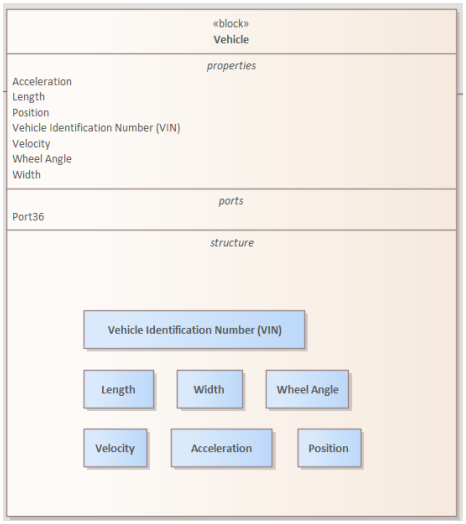

Fig. 9. Vehicle Internal Block Diagram

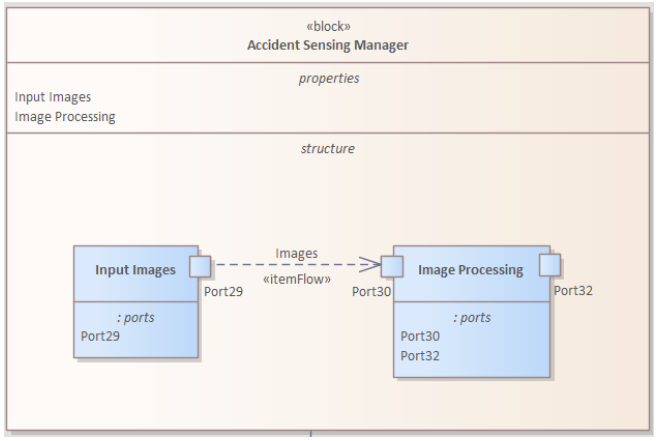

Fig. 10. Accident Sensing Manager Internal Block Diagram

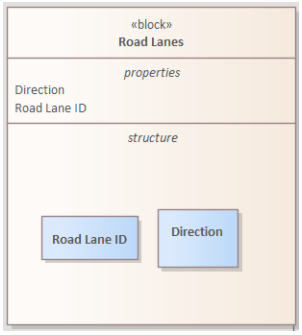

Fig. 11. Road Lanes Internal Block Diagram

C. State Machines

We considered states of 3 components in our project: Intersection Manager, Vehicle, and Accident Sensing Manager.

Intersection Manager (*Fig. 12.*) starts with IDLE state as there is no vehicle on the road. As soon as vehicles approaching the intersection, it gets requests from those vehicles, it goes into Scheduling Vehicles States and Removing Passed Vehicles later. These two states are in loop, because Intersection Manager constantly schedules new vehicles and removes the ones who already passed the intersection.

Vehicle (*Fig. 13.*) is in Driving state before entering intersection area. As soon as it is in the communication range, it regulates its and acceleration according to the values coming form Intersection Manager for that vehicle. The acceleration can be positive or negative, in other terms, the vehicle can speed up or decrease its speed to precisely approach the intersection. After approaching the intersection, the vehicle goes into passing state, and as soon as it passes the intersection the vehicle goes into driving state.

Accident Sensing Manager (*Fig. 14.*) constantly receives input messages and sending those to Image Processing and if the accident is detected it reports that accident, and gets back to input image receiving state again. So, all the states are executed in the loop.

Activity Behaviour (*Fig. 15.*) can be interpreted as Traffic Control Strategy of our project. This also represents our code implementation flow while maintaining the relevance to the state machines

1. The information list is initially empty.
2. Once a vehicle enters the IM unit's communication range, the vehicle's basic information, including its vehicle ID and route ID, is sent to the IM unit. First, the IM unit examines the conflict matrix. If there is no potential conflict at all (as is the case with right-turning vehicles), the IM unit will send a NULL signal to the vehicle indicating that it has safely passed the intersection and will then proceed to step (4). Otherwise, determine the potential conflict routes for the target route. Routes r1, r4, r8, r10, and r11 are examples of potential conflict routes.
3. Find all vehicles that are currently on potential conflict routes by searching the information list. Return to vehicle the maximum possible leaving time. Set maximum passing time to NULL which is the arrive time of the vehicle if there is no vehicle on the conflict route.
4. The vehicle's speed is adjusted in response to feedback from the IM unit. If the feedback signal is not NULL, it should make an extra effort to arrive at the intersection after the maximum pass time. The vehicle should pass through the intersection as quickly as possible but no later than maximum pass time. Finally, the vehicle transmits to the IM unit its estimated time of exit from the intersection.
5. The IM unit adds vehicle data to the information list. Intersection Manager should be cleared of any vehicles that leave the intersection. And algorithm goes back to step (2).

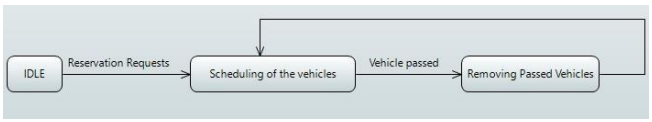

Fig. 12. Intersection Manager State Machine Diagram

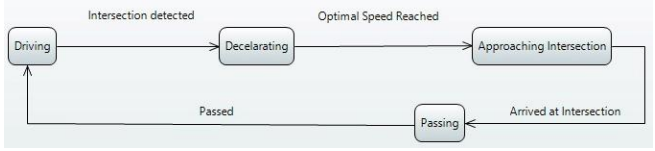

Fig. 13. Vehicle State Machine Diagram

Fig. 14. Accident Sensing Manager State Machine Diagram

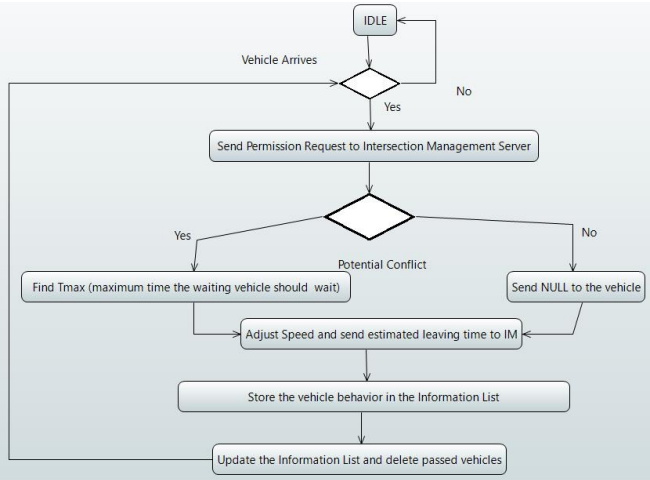

Fig. 15. Traffic Control Strategy Activity Diagram

D. *Constraints*

Constraints are the mathematical equations or set points which are used in our system to calculate the relevant attributes. We are mainly concerned with calculating speed, distance, time (*Fig. 17.*) etc. as well as having fixed values (*Fig. 16.*).

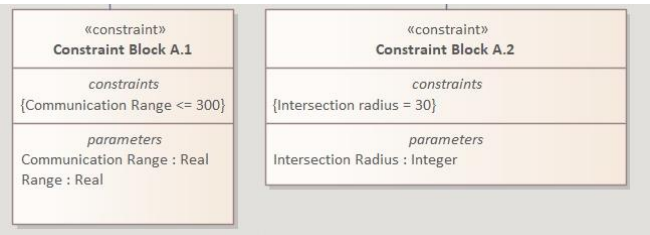

Fig. 16.

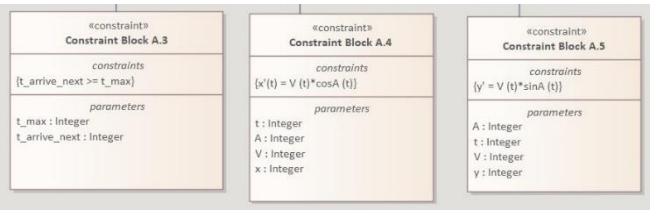

Fig. 17.

IV. IMPLEMENTATION

A. *Algorithm*

The section is aimed to partly code and simulate the state machine behavior. We will use the abovementioned Traffic Control Strategy to achieve this.

We will start with building data models (*Fig. 18.*) for the objects we are going to use in the code. We are going to have Info and Vehicle models, the former is used to create vehicle instance to store data in Information List of Intersection Manager, while the latter is used as Vehicle interface holding actual vehicle attributes. Additionally, we need Intersection Manager model to handle our overall strategy with the functionalities such as adding vehicle, scheduling and removing passed vehicles. In addition, during the calculations Conflict Matrix (*Fig. 19.*) is going to be used. In our case it is a 2D array consisting of ones and zeros indicating which roads are conflicted, and which are not.

Next, to achieve reusability, we need functions (*Fig. 20.*) to calculate vehicle attributes, conflict checks and etc. operations to which are repeatedly used during code flow.

There are several functions regarding speed, distance, time calculations as well as conflict matrix checks and client-service requests.

The main code flow (*Fig. 23.*) consists of creating vehicles, sending requests to Intersection Manager which then calculates their speeds and passing times, returning response, storing the vehicles in Information List, and finally removing passed vehicles form Information List.

```
struct Info
{
  int id;
  int route;
  float passTime;
};

class Vehicle
{
  public:
    int id;
    float speed;
    float length;
    int route;
    Vehicle(int id, float speed, float length);
    float sendRequest();
    void setRoute(int route);
};

class IntersectionManager{
  public:
    IntersectionManager();
    String scheduleVehicle(Vehicle &vehicle);
    void removePassedVehicles(int vehiclesCount);
};
```

Fig. 18. Data Models

```
int conflictMatrix[rows][cols] = {
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1},
    {0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1},
    {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1},
    {0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0},
    {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
    {0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0},
    {0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
    {0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0},
    {0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1}};
```

Fig. 19. Conflict Matrix

```
loat calcSpeed(float distance, float time);
float calcTime(float distance, float speed);
float calcIntersectionInnerDistance(int route);
bool checkConflict(int targetRoute);
bool checkVehicleHasConflict(int targetRoute, int prevVehicleRoute, float prevVehicleRoutePassTime);
```

Fig. 20. Functions

```
IntersectionManager::scheduleVehicle(Vehicle &vehicle)
{

    bool conflict = false;
    float arriveTime = 0;
    float speed = vehicle.speed;
    float passTime;
    // check conflict matrix
    conflict = checkConflict(vehicle.route);

    if (conflict)
    {
      // get pass times of latest vehicle in conflicted route
      for (int j = 0; j < vehicleCount; j++)
      {
        // store the pass time of conflicted vehicle as arrive time of incoming vehicle
        if (checkVehicleHasConflict(vehicle.route, infoList[j].route, infoList[j].passTime))
        {
          arriveTime = infoList[j].passTime; // pass time of previous vehicle
        }
      }


    }

    // calculate arrive time and speed
    speed = calcSpeed(distanceFromIntersection, arriveTime);
    arriveTime = calcTime(distanceFromIntersection - vehicle.length, speed);

    // calculate pass time
    passTime = arriveTime + calcTime(calcIntersectionInnerDistance(vehicle.route) + vehicle.length, speed);

    // return pass time and speed
    return [passTime, speed];
}
```

Fig. 21. Schedule Vehicles

```
IntersectionManager::removePassedVehicles(int vehiclesCount){
      // Start time
      unsigned long startingSeconds;
      int sizeOfInfoList = vehiclesCount;

      while (sizeOfInfoList)
      {
        // find the index of vehicle, the pass time of which is greater than current time
        int vehicleToDeleteIndex = -1;
        for (int i = 0; i < sizeOfInfoList; i++)
        {
            // Current time
          unsigned long currentSeconds = millis() / 1000;

          // If passed time is lower than the difference of current time and start time
          // it means that the vehicle has already passed and can be deleted
          if (infoList[i].passTime <= currentSeconds - startingSeconds)
          {
            vehicleToDeleteIndex = i;
          }
        }

        // There is vehicle that already passed
        if (vehicleToDeleteIndex >= 0)
        {
            // Remove the vehicle from the Information List by the index
            /* .... */
            sizeOfInfoList--;
        }
      }
    }
```

Fig. 22. Remove Passed Vehicles

```
for (int i = 0; i < vehicleCount; i++)
{
  // Create Vehicles
  vehicles[i] = new Vehicle();
}


for (int i = 0; i < vehicleCount; i++)
{
  // Sending request to Intersection Manager and getting calculated passing time as response
  float passTime = vehicles[i]->sendRequest();

  // Store the vehicle in Information List
  infoList[i].id = vehicles[i]->id;
  infoList[i].route = vehicles[i]->route;
  infoList[i].passTime = passTime;
}

// Vehicle's Approaching and Passing State
// Intersection Manager's Remove State
intersectionManager.removePassedVehicles(vehicleCount);
```

Fig. 23. Main Flow

B. *Testing*

We implemented unit tests and defect tests in the project. Unit tests (*Fig. 24.*) are used to test functions, while defect tests (Fig. 26.) are done to find defects in the system and solve them. The unit tests include the testing of the speed, distance, time calculations, conflict matrix checks and vehicle request to Intersection Manager. On the other hand, the defect tests found out the possible defects in removing data from list, conflicted route checks, time calculations, comparison accuracy and data type mismatches. All the functions existing in the code are unit tested:

```
TEST("Should check conflict matrix", testCheckConflict);
TEST("Should calculate speed", testCalcSpeed);
TEST("Should calculate time", testCalcTime);
TEST("Should check vehicle has conflict", testCheckVehicleHasConflict);
TEST("Should calculate intersection inner distance", testCalcIntersectionInnerDistance);
TEST("Should return speed from vehicle's request to Intersection Manager", testVehicleSendRequest);
```

Fig. 24. Unit Tests

Defect: When none of the vehicles's pass time is reached, then the vehicle index will be -1 which causes error when removing from Information List
Solution: The index is checked whether it is greater than or equal to 0.

Defect: Float value equality comparison gives inaccurate results in microcontroller.
Solution: The absolute values are found and the difference should be lower than 0.01 (enough accuracy in our case).

Defect: When the arrive time of vehicle is not important (right turning roads), it becomes 0 and then the speed calculation gives infinity.
Solution: The arrive time is checked whether it is greater than or equal to 0.

Defect: When the vehicle gets the speed and pass time as a response from the Intersection Manager, the values are accepted as string values which caused error in further mathematical calculations.
Solution: The speed and pass time are converted to float.

Defect: When the conflict matrix checked against the routes we were always getting "conflicted" result, because inspite of having conflict with no other road, the road is conflicted with itself.
Solution: The road is checked only against the other routes than itself.

Fig. 25. Defect Tests

```
-------
RUN TEST

PASSED TEST Should check conflict matrix

-------
RUN TEST

PASSED TEST Should calculate speed

-------
RUN TEST

PASSED TEST Should calculate time

-------
RUN TEST

PASSED TEST Should check vehicle has conflict

-------
RUN TEST

PASSED TEST Should calculate intersection inner distance

-------
RUN TEST

PASSED TEST Should return speed from vehicle's request to Intersection Manager
```

Fig. 26. Unit Tests Passed

C. *Simulation*

In simulation part, we executed the project code in TinkerCAD platform via Arduino UNO microcontroller. Initially we create and show the list vehicles with their attributes (Fig. 27.), then we log Information List of the scheduled vehicles with pass times and routes. Finally, vehicles are removed one by one as they pass and the information is also logged in Console.

```
Vehicles' Driving State (Driving)
Vehicle: 1 Route: 0 Speed: 60.00 Length: 3.00
Vehicle: 2 Route: 1 Speed: 60.00 Length: 4.00
Vehicle: 3 Route: 2 Speed: 60.00 Length: 3.00
Vehicle: 4 Route: 3 Speed: 60.00 Length: 4.00
Vehicle: 5 Route: 4 Speed: 60.00 Length: 3.00
Vehicle: 6 Route: 5 Speed: 60.00 Length: 4.00
Vehicle: 7 Route: 6 Speed: 60.00 Length: 3.00
Vehicle: 8 Route: 7 Speed: 60.00 Length: 4.00
Vehicle: 9 Route: 8 Speed: 60.00 Length: 3.00
Vehicle: 10 Route: 9 Speed: 60.00 Length: 4.00
Vehicle: 11 Route: 10 Speed: 60.00 Length: 3.00
Vehicle: 12 Route: 11 Speed: 60.00 Length: 4.00
Vehicle: 13 Route: 0 Speed: 60.00 Length: 3.00
Vehicle: 14 Route: 1 Speed: 60.00 Length: 4.00
Vehicle: 15 Route: 2 Speed: 60.00 Length: 3.00
Vehicle: 16 Route: 3 Speed: 60.00 Length: 4.00
Vehicle: 17 Route: 4 Speed: 60.00 Length: 3.00
Vehicle: 18 Route: 5 Speed: 60.00 Length: 4.00
Vehicle: 19 Route: 6 Speed: 60.00 Length: 3.00
Vehicle: 20 Route: 7 Speed: 60.00 Length: 4.00
```

Fig. 27.

```
Vehicles' Approaching and Passing State
Vehicle: 1 Route: 0 Pass Time: 6.00
Vehicle: 2 Route: 1 Pass Time: 7.00
Vehicle: 3 Route: 2 Pass Time: 6.00
Vehicle: 4 Route: 3 Pass Time: 6.00
Vehicle: 5 Route: 4 Pass Time: 8.00
Vehicle: 6 Route: 5 Pass Time: 8.00
Vehicle: 7 Route: 6 Pass Time: 6.00
Vehicle: 8 Route: 7 Pass Time: 10.00
Vehicle: 9 Route: 8 Pass Time: 10.00
Vehicle: 10 Route: 9 Pass Time: 6.00
Vehicle: 11 Route: 10 Pass Time: 13.00
Vehicle: 12 Route: 11 Pass Time: 12.00
Vehicle: 13 Route: 0 Pass Time: 6.00
Vehicle: 14 Route: 1 Pass Time: 15.00
Vehicle: 15 Route: 2 Pass Time: 15.00
Vehicle: 16 Route: 3 Pass Time: 6.00
Vehicle: 17 Route: 4 Pass Time: 19.00
Vehicle: 18 Route: 5 Pass Time: 18.00
Vehicle: 19 Route: 6 Pass Time: 6.00
Vehicle: 20 Route: 7 Pass Time: 22.00
```

Fig. 28.

```
Intersection Manager's Remove State
Vehicle: 19 removed from vehicle list after 6 seconds
Vehicle: 16 removed from vehicle list after 6 seconds
Vehicle: 13 removed from vehicle list after 6 seconds
Vehicle: 10 removed from vehicle list after 6 seconds
Vehicle: 7 removed from vehicle list after 6 seconds
Vehicle: 4 removed from vehicle list after 6 seconds
Vehicle: 3 removed from vehicle list after 6 seconds
Vehicle: 1 removed from vehicle list after 6 seconds
Vehicle: 2 removed from vehicle list after 7 seconds
Vehicle: 6 removed from vehicle list after 8 seconds
Vehicle: 5 removed from vehicle list after 8 seconds
Vehicle: 9 removed from vehicle list after 10 seconds
Vehicle: 8 removed from vehicle list after 10 seconds
Vehicle: 12 removed from vehicle list after 12 seconds
Vehicle: 11 removed from vehicle list after 13 seconds
Vehicle: 15 removed from vehicle list after 15 seconds
Vehicle: 14 removed from vehicle list after 15 seconds
Vehicle: 18 removed from vehicle list after 18 seconds
Vehicle: 17 removed from vehicle list after 19 seconds
Vehicle: 20 removed from vehicle list after 22 seconds
```

Fig. 29.

D. Scheduling

In this part, we aim to schedule the tasks executed by Intersection Manager based on the execution time of the microcontroller. There are 3 tasks in total:

1. Store vehicle in Information List
2. Get passed vehicle from list.
3. Remove vehicle from the list.

As can be seen from the task execution table that the fastest task is “storing vehicle”, the second fastest is “getting vehicle”, while the slowest task is “removing vehicle”. We try to schedule the tasks based on Rate Monotonic (RM) and Earliest Deadline First (EDF) approaches. Earliest Deadline First (EDF) is a dynamic scheduling algorithm used in real-time operating systems to manage processes and allocate resources. It is a type of scheduling algorithm that prioritizes tasks based on their deadlines. The task with the earliest deadline is given the highest priority and is executed first. On the other hand, Rate Monotonic (RM) is a fixed priority scheduling algorithm that assigns priorities to tasks based on their periodic arrival times, also known as their "rates". The algorithm assigns higher priorities to tasks with shorter periods or faster rates, with the idea that the shorter the period, the more critical the task.

We calculated the total utilization, based on the periods and deadlines assigned by us:

$$U = U_1 + U_2 + U_3 = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} = 0.7796$$

$$U_{RM} = n * \left(2^{\frac{1}{n}} - 1\right) = 0.7798$$

$$U_{EDF} = 1$$

$$U < U_{RM} < U_{EDF}$$

As can be clearly seen from the calculations, the tasks are schedulable both in terms of RM and EDF approaches. The figures below demonstrate individual and merged executions of tasks over 100 milliseconds period.

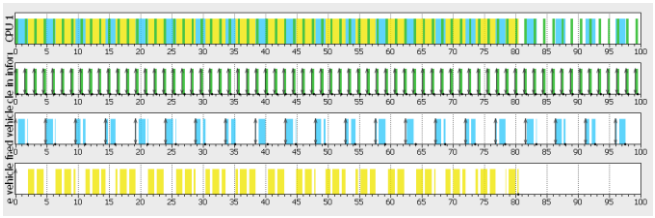

Fig. 16. EDF Scheduling

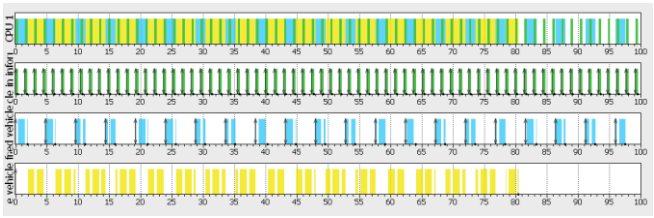

Fig. 17. RM Scheduling

V. Conclusion

By way of conclusion, autonomous intersection management has the ability to transform the way intersections are handled while also reducing the amount of accidents and traffic delays. Autonomous intersections can reliably detect and adapt to changing traffic conditions by utilizing modern technology such as sensors, cameras, and communication systems, increasing traffic flow and lowering the chance of accidents. With the advantages it provides, such as greater safety, decreased overcrowding, and increased efficiency, the future of junction management is bright. The continuing development and implementation of autonomous intersection management technology will be important in influencing mobility and transportation in the future. we propose an intersection management strategy based on vehicle-to-infrastructure communication in this paper. Our strategy keeps a static conflict matrix and a dynamic information list, as well as performing tasks like storing and searching. Despite this, the intersection management strategy proposed in this project is largely heuristic. Our “TinkerCAD” simulations show that our method has high traffic efficiency. To improve the overall efficiency of the traffic network, an IM strategy could systematically consider multiple upstream and downstream intersections. Autonomous and human-operated vehicles will coexist on the road soon. As a result, intersection management that considers this coexistence is required. Another irrefutable improvement would be allowing vehicles more freedom to optimize their decisions (e.g., overtaking and changing lanes) near intersections may improve traffic efficiency even further.

VI. References

[1] Chouhan AP, Banda G. Autonomous intersection management: A heuristic approach. IEEE Access. 2018; 6: 53287–53295.

[2] Cruz-Piris L, Lopez-Carmona MA, Marsa-Maestre I. Automated optimization of intersections using a genetic algorithm IEEE Access. 2019; 7: 15452–15468.

[3] Dresner K, Stong P. Multiagent traffic management: A reservation-based intersection control mechanism. In: Proceedings of the 3rd international joint conference on autonomous agents and multiagent systems (AAMAS), New York, USA, 19-23 July 2004, pp.1-8.

VII. Appendix

Source Code: GitHub Repository

A. Structure of repository

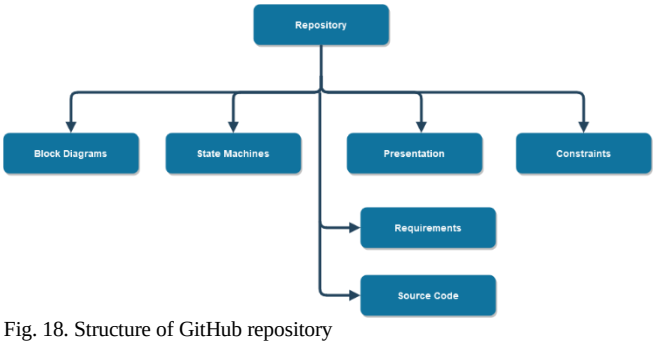

Fig. 18. Structure of GitHub repository

B. Commits

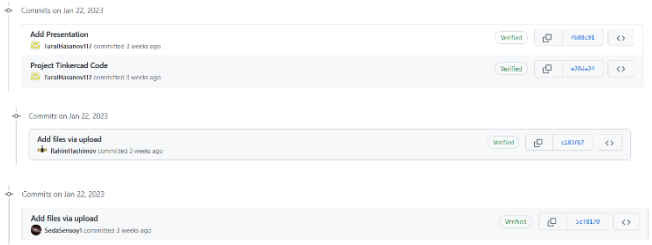

Fig. 19. Git commits.

IEEE χονφερενχε τεμπλατεσ χονταιν γυιδανχε τεξτ φορ χομποσινγ ανδ φορματτινγ χονφερενχε παπερσ. Πλεασε ενσυρε τηατ αλλ τεμπλατε τεξτ ισ ρεμοωεδ φρομ ψουρ χονφερενχε παπερ πριορ το συβμισσιον το τηε χονφερενχε. Φαιλυρε το ρεμοωε τεμπλατε τεξτ φρομ ψουρ παπερ μαψ ρεσυλτ ιν ψουρ παπερ νοτ βεινγ πυ