Network Traffic Controlling System

Vibhashree Vijayendra Hippargi
Dept. of Computer Science
FH Dortmund
Dortmund, Germany
vibhashree.hippargi001@stud.fh-dortmund.de

Supadma Kadabi
Dept. of Computer Science
FH Dortmund
Dortmund, Germany
supadma.kadabi002@stud.fh-dortmund.de

Sachin Kumar
Dept. of Computer Science
FH Dortmund
Dortmund, Germany
sachin.kumar003@stud.fh-dortmund.de

Hamida Aliyeva
Dept. of Computer Science
FH Dortmund
Dortmund, Germany
hamida.aliyeva004@stud.fh-dortmund.de

***Abstract* — This document explains about Network Traffic Controlling Systems that build the Smart City. Enterprise architect(EA) is used for modeling the requirements and scenarios, hardware implementation is done with TinkerCAD, the code implementation for the accident scenario for vehicle is done with C++ and Google test(GTest) is used for testing the scenario.**

Keywords — EA, V2X, TinkerCAD, C++, GTest

I. INTRODUCTION

author - Supadma Kadabi

With the advancement in technology there is a huge shift in the mobility industry. From the development of the first motor car to today's fully autonomous cars, the mobility industry has witnessed extreme development. With such advancements in the mobility industry the infrastructure also needs to be updated for new age cars. This paper proposes one such infrastructure change to update the autonomous car and intersections to be aware fully of the changes in its surroundings. The intersections are proposed to be more than traffic lights to be able to send live information about the conditions of the path to the cars driving in that path. The given system assumes that all the cars available in the system are mostly automated to take decisions related to path change and be able to send messages to each other while traveling. The system concentrates on scenarios such as traffic jam, battery drain out or population concentrated areas and how to tackle the situation.

The system proposes the use of communication technology V2X. Here the vehicles are communicating with various other entities in their environment. In V2X protocol vehicles communicate with other Vehicles, Infrastructure, Pedestrians or Devices. V2X is also described as Cellular V2X or Dedicated Short-Range communication. This protocol improves the fuel usage, avoids accidents and efficiently manages traffic. In this protocol vehicles are connected to each other using WLAN, which reduces the latency. The messages flowing here are known as Cooperative Awareness Messages or Basic Safety Messages and Decentralized Environmental Notification Messages. With the latest advancement in cellular networks now it is known as Cellular V2X. In the recently developed Cars both traditional V2X and Cellular V2X are available. It also supports various levels of support as shown in the following figure:

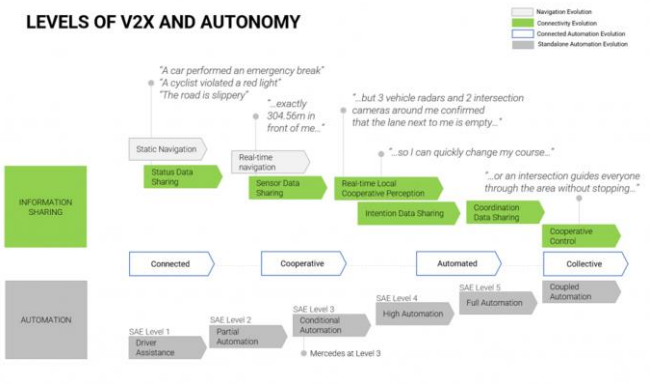


Fig 1: Levels of Autonomy in V2X

The proposed system has various applications as shown in the block diagram. For example, the Smart Path Finder application of this system can be used in various scenarios. It majorly works based on the understanding of the city layout and traffic movement in the particular time of the day. In this example, the system observes the traffic jam in the given path and updates the next car in the same path about the jam. There are other scenarios like when it is the peak hours of the day and the autonomous car has selected a path which involves passing by a school, the system updates the car and suggests a new path. The hospital zones are also marked and the entry and exit points for the ambulance are marked so that Autonomous cars will not disturb the Ambulance path and choose alternate paths. Under some special conditions like Accidents, the autonomous cars are updated with possible next steps based on alternative path availability or change of lane for hassle free commute.

II. REQUIREMENTS MODEL

author - Supadma Kadabi

Requirements are the conditions or needs that must be fulfilled to complete a given project. The engineering that deals with analyzing and understanding the requirements is known as Requirement Engineering. In the Requirement Engineering process multiple steps are followed for successful outcomes. The important steps are Requirements Elicitation, Requirements Analysis, Requirements Specification, Requirements Validation and Requirement Management. In the proposed system the requirements are elicited with multiple group discussion. The specification document has ten requirements which overall explain an infrastructure to support the autonomous vehicles.

The requirements majorly cover five main subtopics:

1. The system shall be able to suggest the autonomous vehicle with alternative paths under circumstances like Traffic jam, School zones, Hospital zones, Accidents.

2. The system shall be able to suggest the autonomous vehicle to reduce the speed to a given speed when there is a pedestrian hotspot available, U-turn is available or Sudden brake applied by the car In Front of it.

3. The system shall be able to ask the autonomous car to change lanes when any emergency vehicles with higher priority are encountered on the same lane.

4. The system shall be able to suggest the available parking spot in the nearby locality to the autonomous car after destination is reached

5. The system shall be able to provide information about the nearby charging points and number of vehicles present in the charging point. It shall be able to direct the autonomous vehicle to the next charging point based on delay.

These main sub-topics further have detailed instructions for each step. One of the use cases i.e. Accident Occurred is explained in detail in the given document.

III. BLOCK DEFINITION DIAGRAM

author - Supadma Kadabi

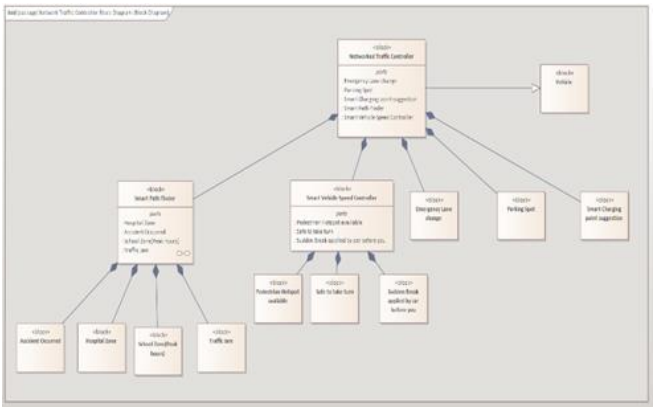
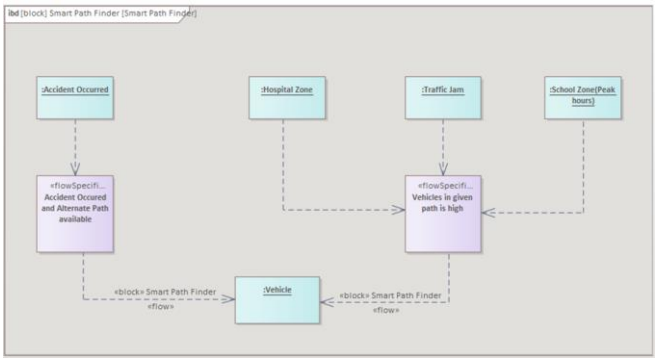


Fig 2: Block Diagram

A Block diagram represents several blocks associated with each other to give a brief understanding of the complete system. Blocks are units of data which contain attributes, operations and constraints. The block definition diagram gives an overall understanding of the system and can be used in debugging by eliminating the blocks based on flow and fault. The blocks can be used in all phases of system specification and design. A block diagram can be used to represent Hardware, Software or Human organizations. In the given block definition diagram, the main system Network Traffic Controller is further connected to several lower-level blocks like Smart Path Finder, Smart Vehicle Speed Controller etc. Each block represents a system which helps Autonomous vehicles to navigate through traffic with less hassle. For example, Smart Charging point suggestion guides the autonomous vehicle about the available parking spots in the nearby Parking space.

IV. INTERNAL BLOCK DIAGRAM

author - Supadma Kadabi



.Fig 3: Internal Block Diagram

As the name suggests Internal block diagram describes the internal schematic of a block. Internal block diagrams can be used to display the ports, parts and connectors. It also can be used to display the sub-system involving multiple blocks. In the given internal block diagram we can see the sub-system for Smart-Path-Finder. We can see that alternate paths have to be found on multiple occasions like Accident occurred, Hospital zones, Traffic jam, School Zone. Under specific circumstances the Network Traffic Controller provides the vehicle with alternate paths for safer and faster commute.

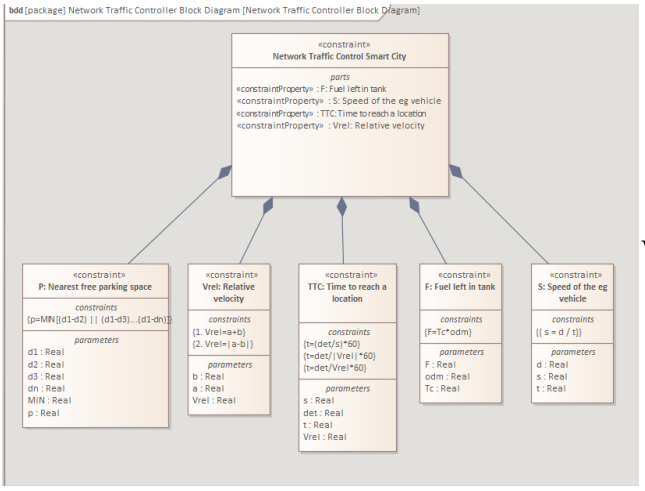
V. CONSTRAINT AND PARAMETRIC DIAGRAMS

author - Vibhashree Hippargi

A constraint diagram represents all the constraints (variables) that must be considered while designing a system. This is important to identify and record the effects of changing one variable would affect which of the blocks. The constraint blocks can be used to specify a group of constraints in the form of a mathematical expression. For Example, Speed = Distance/time.

Every constraint block has 2 parts. Constraints and the parameters. In the above example, a constraint is an expression, whereas distance and time are the parameters.

Representation:
Constraint block contains <<constraint>> tag.
The mathematical expression contains “constraints”.



.Fig 4: Constraint diagram

The parametric diagram shows the relationship between the parameters and the usage of one parameter to derive another constraint. The parametric block contains “Par” in its block name.

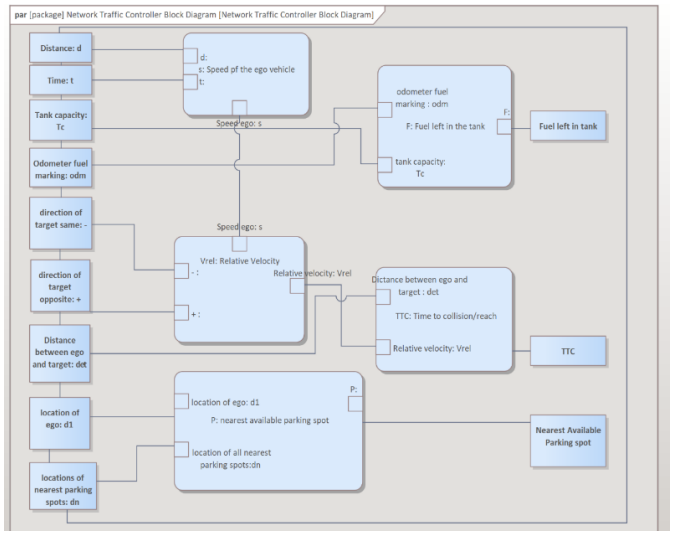


Fig 5: Parametric Diagram

VI. USE CASE

author - Vibhashree Hippargi

Accident Occurred Use case – “Accident Occurred”. When a vehicle is involved in an accident on a particular lane, it sends the message to other approaching road users about the incident via V2. Based on this, the road users either change their lane or change the path if available. By this, not only their time is saved, but also the traffic jam is avoided.

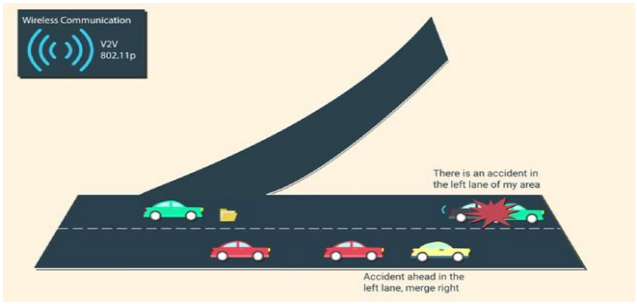


Fig 6: Accident Occurred scenario.

VII. STATE MACHINE FOR THE USE CASE

author - Supadma Kadabi

State Machine Diagrams are used to represent discrete behavior through state transitions. When there is a change in the state there are activities which are performed. These Activities can be continuous or discrete. There are multiple entry and exit criteria which need to be followed. In the given state transition diagram, there are four states: Driving **normal, Change Driving path, Change Driving lane** and **Decelerate**. All the state transitions will follow particular conditions like distance between vehicles is less than 50m, there is an alternate path available for path change etc.

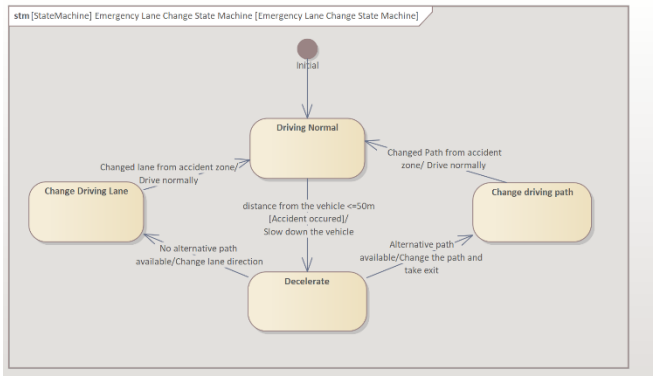


Fig 7: State diagram of Accident Occurred scenario

VIII. IMPLEMENTATION

author - Sachin Kumar

This approach is standard which is efficient and reliable in the Accident Occured Scenario, where the vehicle has to decide what appropriate action to take in case of accident occurrence.

The implementation of the system of Scenario Accident Occured is done by making different states for the vehicle and switching cases to transfer from one state to another state.

The structure of the code is defined using two classes: -
1. Vehicle

```
class Vehicle
{
public:
    Vehicle();
    std::string update_speed(State current);
    std::string drive_normal(State current);
    std::string update_direction(State current, char direction_avail);
    std::string update_lane(State current, char lane_avail);
};
```

Fig 8: *Vehicle* Class Structure

2. Accident_Scenario

```
class Accident_Scenario
{
public:

    int distance_of_ahead_vehicle;
    char accident_occured;
    char alternate_path_available;
    Accident_Scenario();
    Accident_Scenario(int distance_of_ahead_vehicle,
                      char accident_occured,
                      char alternate_path_available);
    void setdistance_of_ahead_vehicle(int dist);
    int getdistance_of_ahead_vehicle();
    void setaccident_occured(char alt);
    char getaccident_occured();
    void setalternate_path_available(char alt);
    char getalternate_path_available();
    State State_Changes(State current);
};
```

Fig 9: *Accident_Scenario* Class Structure

Vehicle class has functions as update_speed, drive_normal, update_direction, and update_lane, while Accident_Scenario class has functions *setdistance_of_ahead_vehicle*, *setdistance_of_ahead_vehicle*, *setaccident_occured*, *getaccident_occured*, *setalternate_path_available*, *getalternate_path_available*, and *State_Changes* functions.

All these functions are responsible for performing tasks which will control the behavior of the robot. For example, *getalternate_path_available* is used to know if the alternate path for the vehicle is available and getaccident_occured is used to find if the accident has occurred on the road or not. *State_Changes* function performs the tasks to switch the States if certain conditions are fulfilled. These conditions depend on various arguments, e.g. *distance_of_ahead_vehicle*, *accident_occur*, *path_available*, and *current_state*.

The state machine management is handled by Switch Cases. The different states for the vehicle are defined in enum STATE and these states are driving_normal, decelerating, changed_path, and changed_lane_direction.

```
enum class State
{
    driving_normal = 1,
    decelerating,
    changed_path,
    changed_lane_direction
};
```

Fig 10: States Enum

In the *main* function the function *function_response* uses the Accident_Scenario *State_Changes function* which uses the Switch function to change the state of the Vehicle and then returns the response.

The scenario explained is about the Car which has to decide, when the accident has occurred, does it need to drive normally, decelerate, change path or change lane direction.

The State_Change function uses the switch case and uses the *current* state variable to identify the state and then execute the logic to change the state.

In case, the vehicle is in *driving_normal* state, the *accident_occured and distance_of_ahead_vehicle* arguments are checked. If the *accident_occured* and *distance_of_ahead_vehicle* is less than 50 units, then the State of vehicle is changed to *decelerating* otherwise it is *driving_normal.*

```
case State::driving_normal:
    if ((accident_occured == 't') && (distance_of_ahead_vehicle <= 50))
    {
        current = State::decelerating;
        cout << "Message: Vehicle decelerate" << endl;
    }
    else
    {
        cout << "Message: Vehicle drive normal" << endl;
    }
    return (current);
    break;
```

Fig 11: *driving_normal* State

In case, the vehicle is in *decelerating* state, the *alternate_path_available* argument is checked. If the *alternate_path_available* is true, then the State of vehicle is changed to *changed_path* else it is changed to *changed_lane_direction.*

```
case State::decelerating:

    usleep(5000);
    if (alternate_path_available == 't')
    {
        current = State::changed_path;
        cout << "Message: Vehicle change path" << endl;
    }
    else
    {
        current = State::changed_lane_direction;
        cout << "Message: Vehicle change lane direction" << endl;
    }
    return (current);
    break;
```

Fig 12: *decelerating* State

In case, the vehicle is in *changed_path* state, the State of vehicle is changed to *driving_normal.*

```
case State::changed_path:

    usleep(5000);
    current = State::driving_normal;
    cout << "Message: Vehicle drive normal" << endl;

    return (current);
    break;
```

Fig 13: *changed_path* State

In case, the vehicle is in *changed_lane_direction* state, the State of vehicle is changed to *driving_normal.*

```
case State::changed_lane_direction:

    usleep(5000);
    current = State::driving_normal;
    cout << "Message: Vehicle drive normal" << endl;

    return (current);
    break;
```

Fig 14: *changed_lane_direction* State

Main function of the code to execute the state machine behavior based on the attributes like*distance_of_ahead_vehicle*, *accident_occur*, *path_available*, and *current_state*. Then call the Accident_Scenario function with required attributes. Next, the *function_response* uses the attributes and returns the response of the Vehicle.

```
int distance_of_ahead_vehicle = 0;
char accident_occur = 'f';
char path_available = 'f';
State current_state = State::driving_normal;
Accident_Scenario a1(distance_of_ahead_vehicle,
                     accident_occur, path_available);

cout << "Press q to Quit, 't' for True and 'f' for False \n";
while (true)
{
    cout << "My distance from message transmitter: ";
    if (!(cin >> distance_of_ahead_vehicle))
    {
        cout << "Integer numbers only allowed";
        break;
    }
    cout << "Enter if Accident occured: ";
    cin >> accident_occur;
    if (accident_occur == 'q')
    {
        break;
    }
    cout << "Enter if Alternate path available: ";
    cin >> path_available;
    if (path_available == 'q')
    {
        break;
    }
    a1.setdistance_of_ahead_vehicle(distance_of_ahead_vehicle);
    a1.setalternate_path_available(path_available);
    a1.setaccident_occured(accident_occur);

}
string response = function_response(distance_of_ahead_vehicle,
                    accident_occur, path_available, current_state);

cout << response;
```

Fig 15: *main* function

The arguments *distance_of_ahead_vehicle*, *accident_occur*, *path_available*, and *current_state* are taken from the user through command line arguments.

```
Press q to Quit, 't' for True and 'f' for False
My distance from message transmitter: 25
Enter if Accident occured: t
Enter if Alternate path available: t
Message: Vehicle decelerate
Acknowledgement: Vehicle decelerating
Message: Vehicle change path
Acknowledgement: Vehicle Changed Path
Message: Vehicle drive normal
Acknowledgement: Vehicle drive normal
```

Fig 16: Accident Scenario Path Change

In this scenario, the accident occurred is true and the alternative path available is also true. Then the Vehicle decelerates, then changes the path and drives normally.

```
Press q to Quit, 't' for True and 'f' for False
My distance from message transmitter: 50
Enter if Accident occured: t
Enter if Alternate path available: f
Message: Vehicle decelerate
Acknowledgement: Vehicle decelerating
Message: Vehicle change lane direction
Acknowledgement: Vehicle Changed Lane Direction
Message: Vehicle drive normal
Acknowledgement: Vehicle drive normal
```

Fig 17: Accident Scenario Lane Change

In this scenario, the accident occurred is true and the alternative path available is false. Then the Vehicle decelerates, then changes the lane direction and drives normally.

IX. TINKER CAD

author - Vibhashree Hippargi

TinkerCad is a web-based 3D modeling tool with various hardware components that may be required to build a working system, all online. No hassle of gathering the real hardware equipment. It also shows the reason for failure when a component blows up, be it because of increased current or other reasons (which will be shown on mouse hover over the component).

Working:

- 4 leds represent 4 as explained below.

Green – Driving Normal
Red – Decelerating
Orange – Changed Path
Blue – Changed Lane direction

- In order to calculate the distance between the accident spot and the ego vehicle, an ultrasonic distance sensor is used.
- 2 push buttons are used to set accident occurred and is an alternate path available to true or false as user inputs.
- The accident occurred is left most push button with the Red LED linked to it. Alternate path available is the next push button with an orange LED linked to it.
- In case the accident occurred is true and the distance is less than the threshold set and there is no alternate path available, then the states change from driving normal to decelerating to changed lane direction and back to driving normal.

Driving normal > Decelerating > Changed Lane direction > Driving Normal

- In case the accident occurred is true and the distance is less than the threshold set and there is an alternate path available, then the states change from driving normal to decelerating to changed path and back to driving normal.

Driving normal > Decelerating > Changed Path > Driving Normal

The figure below shows the implementation of the state machines in TinkerCad.

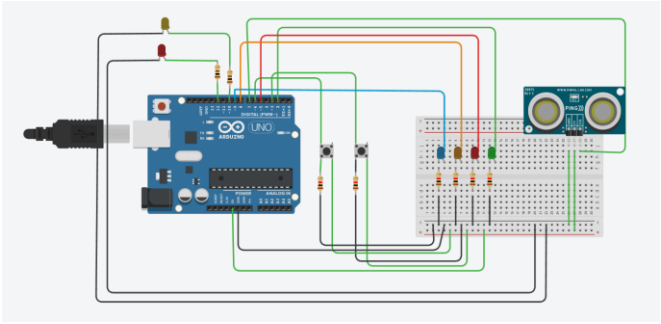


Fig 18: State machine implementation in TinkerCad

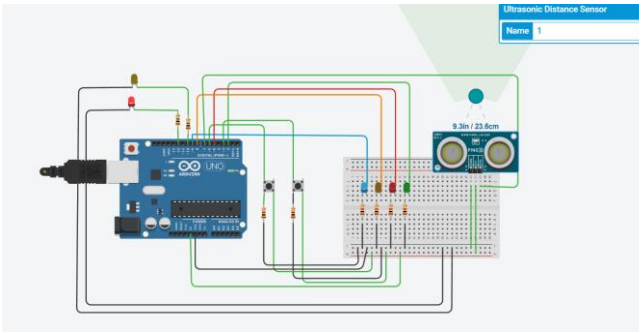


Fig 19: Changed Lane scenario when accident occurred, distance is less than threshold(50) and alternate path NOT available

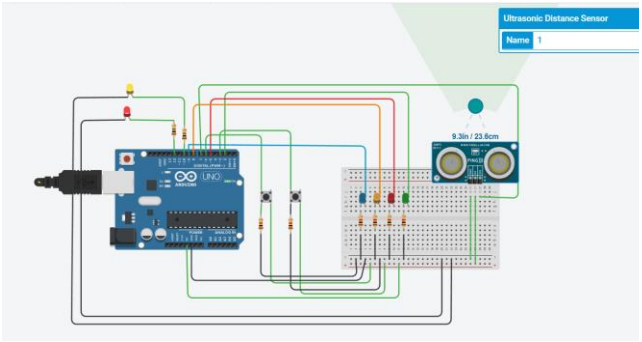


Fig 20: Changed Path scenario when accident occurred, distance is less than threshold(50) and alternate path available

X. SCHEDULING

author - Vibhashree Hippargi

When there exists multiple jobs in a system, scheduling plays a significant role in optimizing the system so that its efficiency and reliability is not compromised. In simple terms, we analyze schedulability to determine whether a set of jobs/tasks are being executed within its deadline or not. In Hard-real time systems where timing is very crucial, scheduling has to be proper. In case any task misses its deadline, it may lead to catastrophic events. Example: a computer system controlling a railway crossing.

Whereas, in Soft-Real time systems, missing the deadline may not cause catastrophic events but it would affect the overall system performance and utility of the result. Example: letter sorting machine.

In this project we are comparing two types of scheduling techniques namely EDF (Earliest deadline first) and RM (Rate Monotonic). To measure the time taken by each component or a function in code, we use the millis() and high resolution clock in c++ respectively. This gives us the approximate time taken to finish a task and we set the deadlines based on their priority.

The formulas needed to verify if the tasks are safe to be scheduled by a particular technique are as follows. Here U is the utilization bound and n are the number of tasks.

• EDF (Earliest Deadline First)
Schedulable iff U <=1

•RM (Rate Monotonic)
Schedulable iff $U <= n(2^{1/n} - 1)$, where n is the number of tasks

Where,
$U = \sum_{\tau_i \in T} \frac{C_i}{T_i}$

n	$U_{lub}(RM, n)$	n	$U_{lub}(RM, n)$
2	0.828	3	0.779
4	0.756	5	0.743
6	0.734	7	0.728
8	0.724	9	0.720
10	0.717	∞	0.693 = ln(2)

Fig 21: Snapshot of formulas for EDF and RM

To get a clear understanding of what the impact would be when EDF or RM is used for scheduling, we consider 3 use cases of the smart city scenario as mentioned below with priorities taken into consideration.

Tasks / Time in ms	Accident Occurred (AO)	Emergency Vehicle approaching (EV)	Pedestrian Hotspot ahead (PH)
Ci (Execution time)	1	2	3
Ti (Period)	4	6	8
Di(deadline)	4	6	8

U = 1/4 + 2/6 + 3/8 = 0.95

Fig 22: Tasks and their deadlines

Using a tool Simso, the Gantt chart of a selected scheduler can be analyzed. The details and snapshots from the same are as explained below.

When RM is used U (0.95) NOT <=0.77. Hence it is NOT safe to schedule in RM.From the image, we can clearly see that Task 3 i.e. Pedestrian Hotspot ahead has missed the deadline. In Hard-RTOS, it would cause severe impacts.

The below figure shows how a tasks misses its deadline.

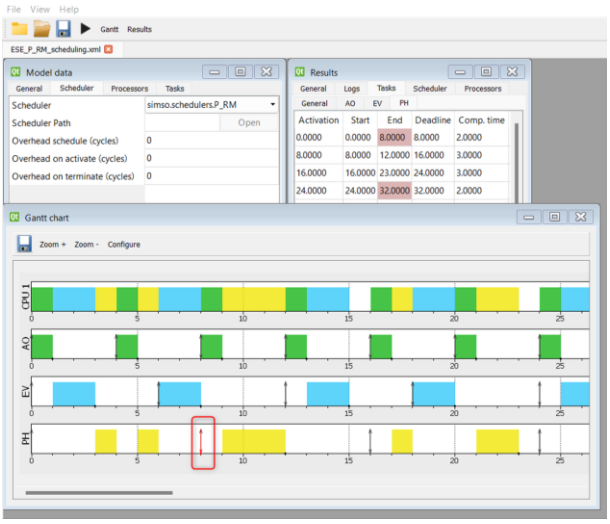


Fig 23: Task3 i.e Pedestrian Hotspot misses its deadline

Whereas, using EDF where U (0.95) is <=1. Hence it can be scheduled in EDF.From the image, we can see that no task has missed its deadline.

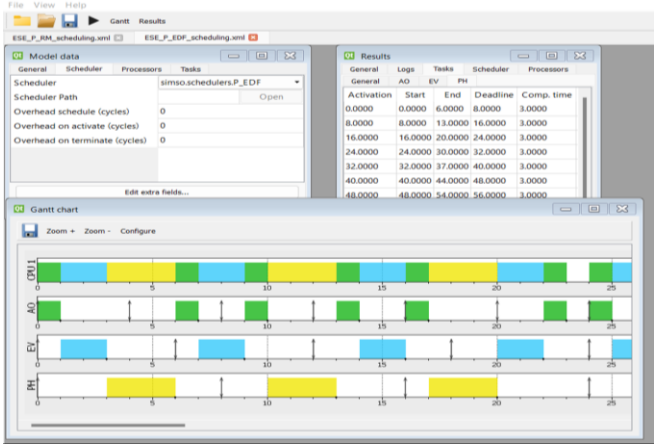


Fig 24: No tasks misses its deadline

Hence by this we conclude that it is safe to schedule tasks using EDF.

XI. INSPECTION

author - Hamida Aliyeva

Inspection, in any field, plays a significant factor in the success of a product. During this phase, potential errors could be identified, and the shortcomings can be overcome in a new release. It also saves the project time and money caused by late detection of bugs. Below listed are the original and improved versions that lead to our final documents:

1. State Machine code has been improved
2. Diagrams have been improved
3. Test cases were reviewed before the execution
4. Push buttons were added to Tinker Cad

XII. TESTING

author - Hamida Aliyeva

Software testing is a technique to determine whether the actual software product complies with expectations and to guarantee that the software is defect-free. Software testing is crucial because it allows any faults or errors in the software to be found early and fixed before the software product is delivered. Reliability, security, and high performance are all ensured by thoroughly tested software, which also leads to time savings, cost effectiveness, and customer pleasure. Hence we wrote unit, defect and interface tests in order to make our application work as expected. We wrote our test with Google Test which is a C++ testing program maintained by Google. To integrate Google test to a C++ project it's required to install the *Test Adaptor for Google Test* package in Microsoft Visual Studio. After that it's possible to import the *gtest* library and start testing.

Unit testing

Software testing with a focus on individual software system units or components is known as unit testing. Unit testing checks that each piece of software operates as intended and complies with specifications.

In the picture below you can see how we wrote our unit tests. We gave different inputs to test if the system would give the predictive response. The main objective here was to test if the system reacts correctly when an accident occurs. The steps and the test results are given in the below pictures.

```
TEST(Alternative_path_not_available_state, State_test) {
    Accident_Scenario a1(10, 't', 'f');
    Vehicle v1;
    State  str1 = a1.State_Changes(State::decelerating);
    EXPECT_EQ(State::changed_lane_direction, str1);
}

TEST(Alternative_path_available_state, State_test) {
    Accident_Scenario a1(10, 't', 't');
    Vehicle v1;
    State  str1 = a1.State_Changes(State::decelerating);
    EXPECT_EQ(State::changed_path, str1);
}

TEST(Get_Accident_occur, State_test) {
    Accident_Scenario a1(10, 't', 'f');
    char i1 = a1.getaccident_occured();
    EXPECT_EQ('t', i1);
}
```

Fig 25: Unit test code example

Once we run the test case, we should see if it fails or passes. In the below picture you can see the results of the unit tests we wrote for the application. As it's seen from Fig 14 the test cases passed successfully. So, we made sure that the system takes corresponding actions when an accident occurs.

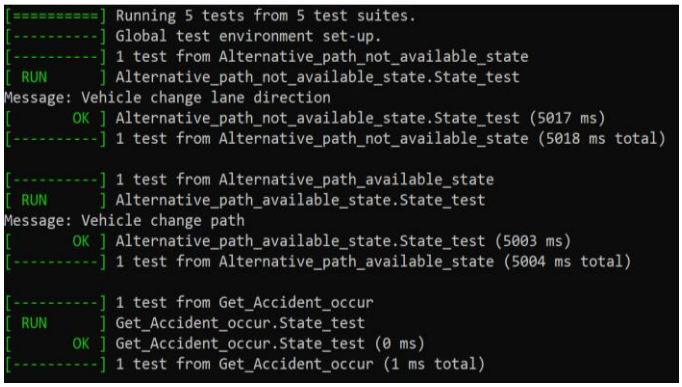


Fig 26: Unit tests successful execution

In general, there are 4 states in our project. In the next test case, we designed unit tests so that they could check each state and verify that they change correspondingly with the input variables. For example, in the first test case below you can see that we have inputs for Accident Scenario. According to the given input we expect the output to be in a decelerating state. EXCEPT_EQ(expected, actual) is used here to check if the expected value of the function is equal to the actual one. As it's seen that the test case passes, hence we can tell the application is working as expected when we enter the inputs above.

Finally, we have 5 unit tests that test different situations and ensure that the functionality of the system works as expected.

Defect test

Defect tests are similar to unit tase but here the goal is to have negative inputs and monitor the system's behavior under unexpected situations. We managed to do it by giving negative inputs and seeing what the system returns as a result. At first, some of our test cases failed as we didn't take every scenario into consideration. After using negative inputs in our tests we discovered those loopholes. This helped us to improve our code as later we added conditions where the system kept working even if the inputs were negative. Below picture is a piece of defect tests that we wrote.

```
TEST(Negative_distance, State_Test) {
    Accident_Scenario a1(-20, 't', 't');
    int i1 = a1.getdistance_of_ahead_vehicle();
    EXPECT_EQ(0, i1);
}
TEST(Negative_accident_input, State_Test) {
    Accident_Scenario a1(20, 'y', 't');
    char i1 = a1.getaccident_occured();
    EXPECT_EQ('f', i1);
}
TEST(Negative_path_input, State_Test) {
    Accident_Scenario a1(20, 't', 'y');
    char i1 = a1.getalternate_path_available();
    EXPECT_EQ('f', i1);
}
```

Fig 27: Defect test code example

When running the test cases we were able to see whether they passed or not. Below in the image you can see that tests are passing as all the required conditions are taken into account..

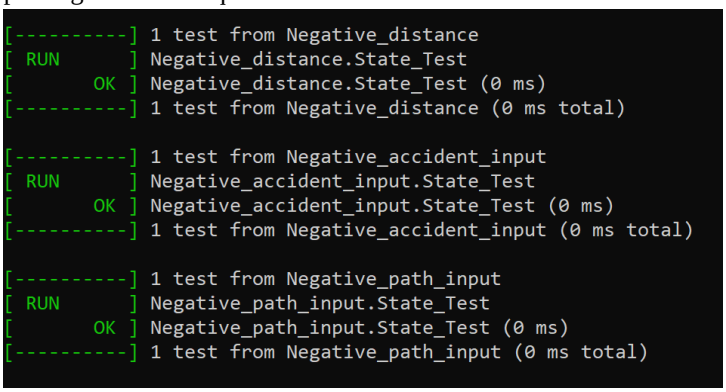


Fig 28: Defect tests execution

We have some strict limits on the values the system can accept. It always first checks if it's less than 50m to the accident, then checks if there's an alternative path available and if not changes the road line. In the first test, we test the output when distance from the accident is a negative value. If this occurs, we should prevent failures. The result of the test case shows that we managed to do it. As in this case the system will continue to work. In addition, to check whether an alternative path is available or not, the system only accepts t as true, and f as false. So, no other letter must be an input. In the test case shown in Fig 15, it's visible that we tested this feature as well and the result in Fig 16 shows that the system will continue to behave normally.

Finally, we have 5 defect tests that are meant to test the system with negative inputs and guarantee that the system's functionality continues to operate as expected.

Interface Test

A sort of software testing technique called interface testing serves as a conduit for interaction or communication between two distant software systems. In our case, we did our interface tests between two classes that we had: Accident_scenario and State. The main idea behind these steps is according to Accident_scenario inputs and State Vehicle must respond with the corresponding messages.. Function_response is the function that creates communication between two classes. We wrote steps where it tested the response of Vehicle according to the output of the Accident_scneario and State. Hence we were able to test interactions between two classes.

In the picture below, you can see the code structure we followed in order to write our tests. We tested one message of the Vehicle by giving one same input to the Accitdent_scenario and three different inputs to the States.

```
TEST(Vehicle_decelerate, Interface_tests) {
    string expected_response = function_response(10, 't', 't', State::driving_normal);
    string actual_response = "Acknowledgement: Vehicle drive normal\n";
    EXPECT_EQ(expected_response, actual_response);
}

TEST(Vehicle_drive_normal, Interface_tests) {
    string expected_response = function_response(10, 't', 'f', State::changed_lane_direction);
    string actual_response = "Acknowledgement: Vehicle drive normal\n";
    EXPECT_EQ(expected_response, actual_response);
}

TEST(Vehicle_change_path, Interface_tests) {
    string expected_response = function_response(10, 't', 't', State::decelerating);
    string actual_response = "Acknowledgement: Vehicle drive normal\n";
    EXPECT_EQ(expected_response, actual_response);
}
```

Fig 29: Interface tests code example

When we run the interface tests we can see that every time the state changes the vehicle's message changes accordingly. The results of the tests are shown in the below picture.

```
[==========] Running 3 tests from 3 test suites.
[----------] Global test environment set-up.
[----------] 1 test from Vehicle_decelerate
[ RUN      ] Vehicle_decelerate.Interface_tests
Message: Vehicle decelerate
Message: Vehicle change path
Message: Vehicle drive normal
[       OK ] Vehicle_decelerate.Interface_tests (10021 ms)
[----------] 1 test from Vehicle_decelerate (10022 ms total)

[----------] 1 test from Vehicle_drive_normal
[ RUN      ] Vehicle_drive_normal.Interface_tests
Message: Vehicle drive normal
[       OK ] Vehicle_drive_normal.Interface_tests (5007 ms)
[----------] 1 test from Vehicle_drive_normal (5008 ms total)

[----------] 1 test from Vehicle_change_path
[ RUN      ] Vehicle_change_path.Interface_tests
Message: Vehicle change path
Message: Vehicle drive normal
[       OK ] Vehicle_change_path.Interface_tests (10022 ms)
[----------] 1 test from Vehicle_change_path (10023 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 3 test suites ran. (25061 ms total)
[  PASSED  ] 3 tests.
```

Fig 30: Interface tests execution

Finally, 3 interface tests are provided to test the functionality of the interface between classes in various scenarios.Based on the test results, we can confidently say that Vehicle produces the expected messages according to the outputs of Actident_scneario and State classes.

XIII. CONCLUSION

author - Supadma Kadabi

The proposed system successfully developed a block definition diagram, internal block diagram, state machine, use case diagram for complete system. It was successful to implement and test the accident occurred scenario where autonomous cars are instructed to change path or lane based on availability.

CHALLENGES:

author - Supadma Kadabi

- This system has assumed all the vehicles to be autonomous, so the compatibility with other semi-automated vehicles or manual vehicles need to be considered.
- The V2X system broadcasts all the messages, the security of the system needs to be strengthened.
- The broadcast of messages may increase the traffic on the channel, message filtering and target selection needs to be added.

XIV. REFERENCES

1) H. Zhou, T. Ma, Y. Xu, N. Cheng and X. Yan, "Software-Defined Multi-Mode Access Management in Cellular V2X," in Journal of Communications and Information Networks, vol. 6, no. 3, pp. 224-236, Sept. 2021, doi: 10.23919/JCIN.2021.9549119.
2) A. Raschke, "Translation of UML 2 Activity Diagrams into Finite State Machines for Model Checking," 2009 35th Euromicro Conference on Software Engineering and Advanced Applications, Patras, Greece, 2009, pp. 149-154, doi: 10.1109/SEAA.2009.60.
3) https://www.commsignia.com/blog/how-v2x-solutions-work/

XV. ACKNOWLEDGMENT

We would like to thank **Prof. Dr. Stefan Henkler** for his lecture and inputs during the exercise session which were helpful in understanding the concepts about the Embedded Systems.

XVI. APPENDIX

Git:
Main branch - used throughout the development)
https://github.com/VibhaHippargi/ESE_SmartCity/tree/main

Final branch – contains relevant files only
https://github.com/VibhaHippargi/ESE_SmartCity/tree/Final

Tinkercad:
https://www.tinkercad.com/things/bVgn3KlB4Fw?sharecode=xnIgK360KLeYYH-lOFKQRjDNJV30t0tzFYQwTk_7K0Q

Contribution % of all team members :

Vibhashree Hippargi – 25%
Sachin Kumar – 25%
Supadma Kadabi – 25%
Hamida Aliyeva – 25%