Embedded Electronic Engineering Report

Andrew Zakhary
Special Emphasis A
Hochschule Hamm Lippstadt
Lippstadt, Germany
Andrew-antwan-mikheal-fahmy.zakhary@stud.hshl.de

Bhavesh
Special Emphasis A
Hochschule Hamm Lippstadt
Lippstadt, Germany
bhavesh@stud.hshl.de

Shahzaib Waseem
Special Emphasis A
Hochschule Hamm Lippstadt
Lippstadt, Germany
shahzaib.waseem@stud.hshl.de

***Abstract*—This document will serve as a technical documentation of this team in order to present their implementation of a real-time cross traffic system, that is able to allow autonomous vehicles to freely travel through an intersection through usage of communication with servers, which detect if a vehicle is able to collide on it's path. The implementation will be split between a software and hardware component. This paper will take the vehicle itself on hardware, whilst the scheduling algorithm for crossing intersections, alongside servers will be implemented on the software level. Uppaal [1] is used to showcase the behaviour of the 4 lanes, with each lane being an individual server. freeRTOS is used in conjunction with the C programming language to implement the queuing algorithm. Vehicle To Infrastructure (V2I) will be used as our main form of deployment for our algorithm, vehicles will communicate with the server and the server will instruct them on their behaviours.**

***Index Terms*—Queue Algorithm, Cross-Traffic Scheduling, Intersection Management, V2X Traffic**

I. Introduction

In real world applications, several times intersections have been congested with traffic whilst traffic signals work hard to make sure that the minimum amount of vehicles keep waiting at each lane. With the usage of V2X, we are able to make this communication between traffic signals and drivers, alot more efficient than a system with human input could be. Autonomous vehicles are able to make independent decisions based off of the data they receive, they do not hold biases such as wanting to accelerate through a corner fast as possible to save a few seconds, but costing the other traffic alot more time. This is where scheduling algorithms can be used to optimised the flow of traffic through an intersection, the software can be directly realised via V2I; Vehicles that communicate with infrastructure are specifically instructed on what to do, and as the vehicle is autonomous with no human input, it is able to realise this request as intended, rarely straying away from the functionality that it was programmed to follow.

This paper will explore how a cross-traffic management system can designed with a simple First In First Out (FIFO) algorithm, allowing vehicles to only pass in the case that a collision is not detected, as an additional safety, accounting for scenarios where a collision is detected, to ensure that it does not continue to accelerate into the other vehicle.

II. Concept

In order to start the design of our system, we first need to understand the system we are creating, for this United Modelling Language (UML) diagrams are used in order to create an abstract definition of our system. This section will discuss these diagrams.

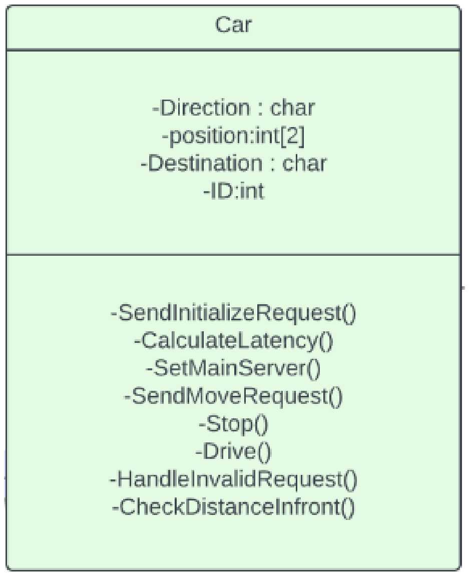


Fig. 1. Class Diagram for autonomous vehicles

A. Class Diagrams

In Fig. 1, a class of Car is created, the attributes and methods shown within the figure were conceptualised based off how the vehicle would interact with the surrounding infrastructure, based off of the input data it receives from the server, it can either Stop() or Drive(). The most crucial information that the server will gather from the vehicle is the Direction it is headed towards.

Fig. 2 is the class diagram with which the class Car will communicates, it is the class diagram of the server. The server is responsible for the decision making process of handling the vehicles coming into the intersection. It works in conjunction with a queue, storing information regarding which vehicle is coming in first. The server can also make the decision to send two vehicles at the same time, if they are not colliding.

Fig. 3 is the class diagram of the queue, it exchanges communication only with the class Server, the data stored here can be read by the different servers in order to determine which vehicle is allowed to pass, there are also flags present in

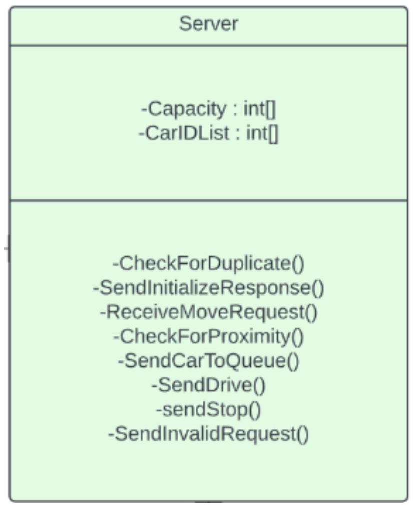


Fig. 2. Class Diagram for Servers

order to determine if a lane is being fully blocked or not. It is important to distinguish if a lane is available, this will led us into the decision-making process of whether a vehicle should be able to pass simultaneously alongside another vehicle or not.

B. Activity Diagrams

Fig. 4 showcases the behaviour of individual vehicles, depending on the stimulus they receive via V2I or V2V, they have to stop if a vehicle is detected within close proximity of them, or if their target lane is busy right now and not able to receive vehicles. The vehicle is then allowed to drive when it receives a request from the server, informing it that no collisions are detected, it is able to reach it's target lane.

Fig. 5 is an activity diagram showcasing how a server would respond in the case that a vehicle is initialised on it. Within our concept phase, we wanted to introduce a check in the form of

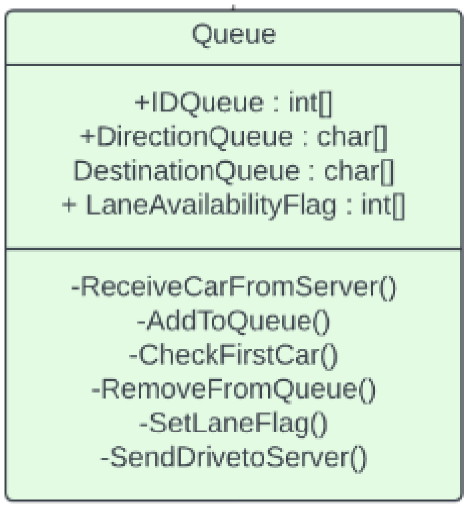


Fig. 3. Class Diagram of the Queue

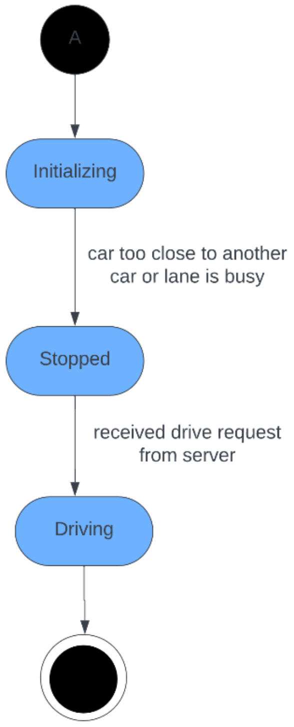

A
Initializing
car too close to another car or lane is busy
Stopped
received drive request from server
Driving

Fig. 4. Behaviour of vehicles

" Is car duplicated ", where it would check for if the vehicle is already within the queue. " Car Inside/Outside Intersection " indicates whether a car has reached the part of the intersection where it has to stop, in order to be processed within the queue, if it has yet to reach that part of the intersection, it is stopped. The activity finally concludes with updating the lane that the vehicle has blocked.

Fig. 6 shows the activity of the queue itself, where it adds a vehicle to queue if there is none present, it then checks the first vehicle within the queue, sending a drive request to it, after which it updates the queue to set lanes as busy, promptly removing that car from the queue.

C. Prototype Application

For our application, we will consider that we have 4 lanes, each having having an entry and exit point. We want to be able to introduce parallelism much as possible within our design. Vehicles should not be constrained to waiting for long times upon entering the intersection, that's why it's important to update blocked lanes and detecting if collisions are occurring. This will help to ensure that our system is safe, in addition to being able to process many cars at the same time through the intersection.

III. Project Management

The team decided to work in an agile manner, as we were receiving new input from our other seminars, getting to know about scheduling algorithms, in addition to studying about hardware/software codesign; As new input was received, we

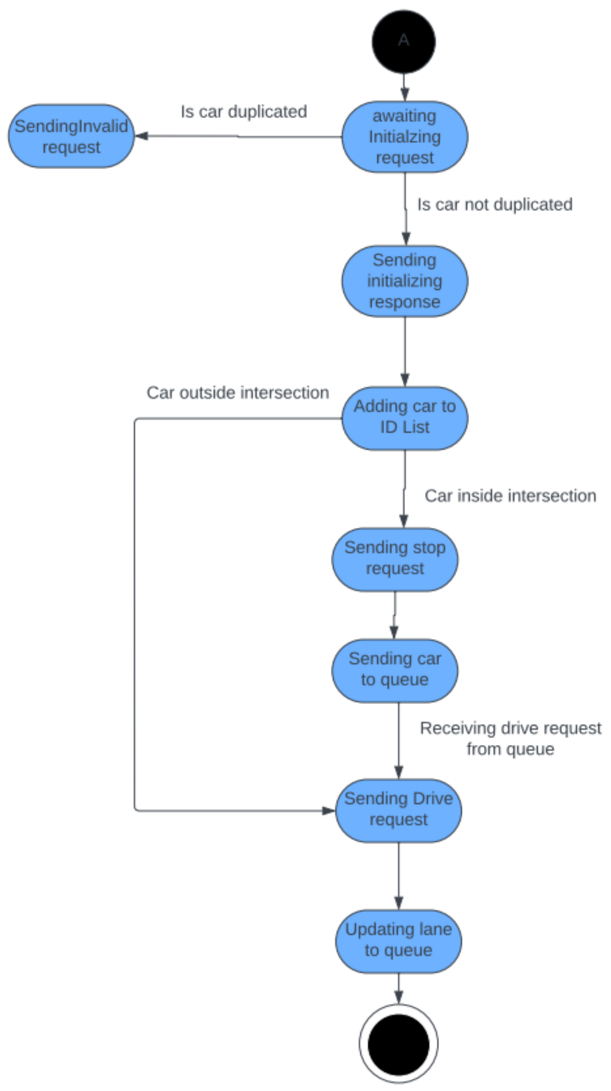


Fig. 5. Behaviour of Server handling a vehicle

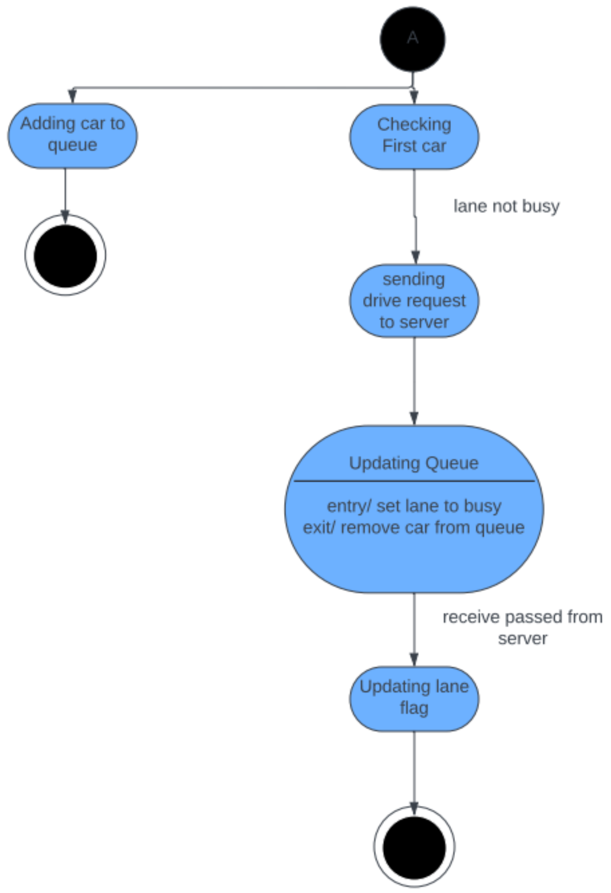


Fig. 6. Behaviour of Server adding to queue and updating lane flags.

wrote up weekly tasks and scheduled weekly meetings to ensure that some progress on the project was being made.

A. Roles and Tasks

There were no set roles within our team, instead everyone worked on all parts of the project, from the conceptualisation to the implementation, we all had to study Uppaal in addition to freeRTOS in order to implement the project. Discussions were highly encouraged within our weekly meetings to criticise and iterate on our existing design.

Some of the tasks that individuals focused on intensively, can generally be broken down as the following:

- **Andrew:** Uppaal Implementation
- **Bhavesh:** Documentation
- **Shahzaib:** freeRTOS Implementation

It is important to emphasise again, that everyone did work together and nobody was handling all the tasks by themselves.

IV. Implementation

This section will now discuss the implementation phase of the project. From our concept phase, we learned alot about the behavioural and structural interaction of the system, however within our implementation phase, a few changes were carried out, such as the decision to not check for proximity with another vehicle when the vehicle has not even initialised at the intersection.

Every vehicle that we add to our queue is the first vehicle to reach that specific lanes entry point.

A. Cars Intersecting

Table I showcases if your vehicle is entering from a row lane X, and then exiting through column lane Y, which lane it ends up having to block. For example, if your vehicle is entering from A, going to C, it will end up blocking B.

This table is important for us in order to not only optimise our code for scenarios that our intersection will encounter, but to also map the behaviour of cars colliding to our programming language.

B. Uppaal

Fig. 7 showcases the behaviour of what Lane A will do, once it receives updates from the server regarding whether it

T / S	1	2	3	4
1	0	[ ]	[2]	[2,3]
2	[3,4]	0	[ ]	[3]
3	[4]	[1,4]	0	[ ]
4	[ ]	[1]	[1,2]	0

TABLE I
ROWS ARE CURRENT LANE, COLUMNS ARE TARGET LANE.

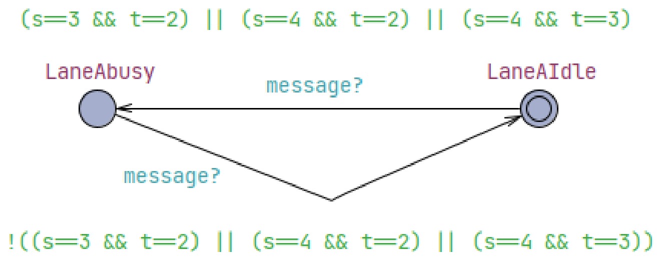

(s==3 && t==2) || (s==4 && t==2) || (s==4 && t==3)
LaneAbusy
message?
LaneAIdle
message?
!((s==3 && t==2) || (s==4 && t==2) || (s==4 && t==3))

Fig. 7. Lane A queuing behaviour

is busy. The information communication of whether the lane is busy or not, is done with the synchronised channel " message ". S is the source of the vehicle, T being the target of the vehicle, the code in green is the optimisation derived from Table I. These guard conditions simulate the behaviour of the lane and what will occur to it, under certain scenarios.

C. *freeRTOS*

freeRTOS is an open-source kernel [2], that is realised within an Arduino [3] library for our application, it simulates how a real-time operating system would work. freeRTOS allows us to simulate the behaviour of vehicles approaching an intersection and subsequently being processed on simulated timing, as this behaviour happens in matters of milliseconds, it can be seen as behaviour that occurs simultaneously rather than one at a time.

In order to initialise our vehicles for the simulation, a Car struct was defined, shown in Fig. 8, this holds within it parameters relevant to our application, those being:

- **ID:** Unique identification of the vehicle, this will be used in order to seperate the different cars that will interact with the intersection.
- **currentLane:** Within Uppaal, it was shown as " S " for source, this will assign a lane to which the vehicle will start from.
- **targetLane:** Within Uppaal, it was shown as " T " for target, this will assign a lane to which the vehicle wants to travel to.

With the usage of this struct, a queue was created in Fig. 9 and cars were initialised as Tasks within the context of freeRTOS. For the purposes of our simulation, we are using

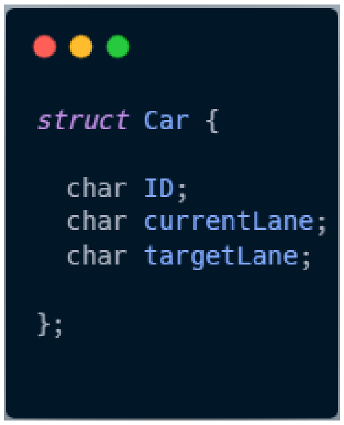

```
struct Car {

   char ID;
   char currentLane;
   char targetLane;

};
```

Fig. 8. Car Struct to simulate vehicles.

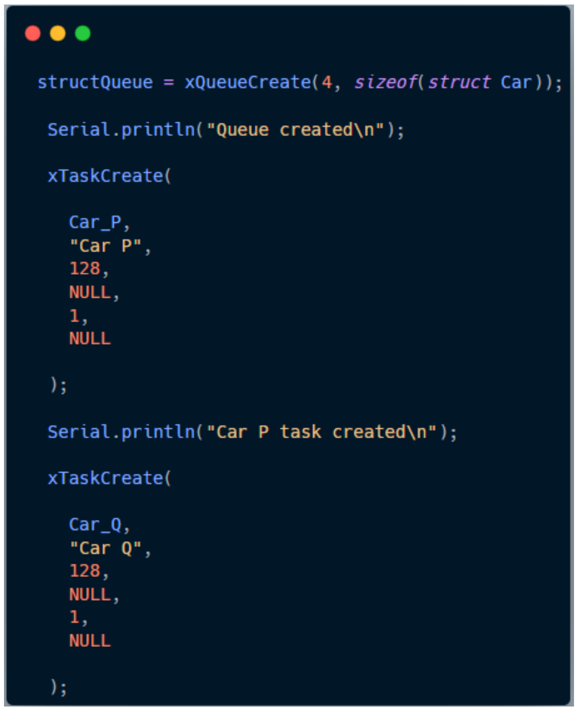
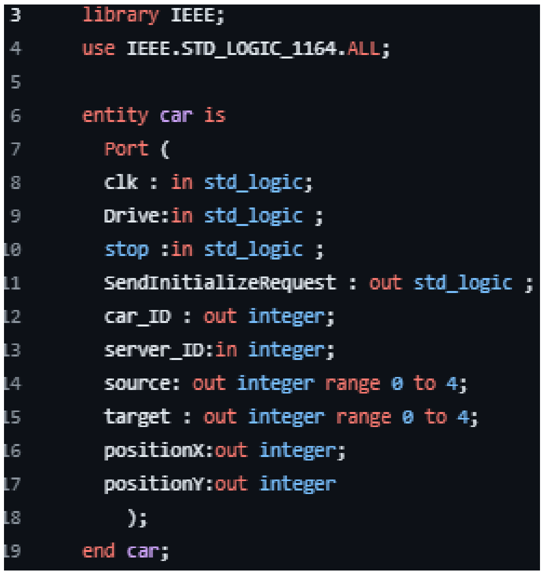

```
structQueue = xQueueCreate(4, sizeof(struct Car));

 Serial.println("Queue created\n");

 xTaskCreate(

   Car_P,
   "Car P",
   128,
   NULL,
   1,
   NULL

 );

 Serial.println("Car P task created\n");

 xTaskCreate(

   Car_Q,
   "Car Q",
   128,
   NULL,
   1,
   NULL

 );
```

Fig. 9. Queue and Cars initialised

4 cars and testing different scenarios by manipulating their currentLane and targetLane attributes.

D. *VHDL*

In order to realise the car and server within our project, we programmed both parts of the project using VHDL [4]. VHDL was selected as the basis of implementing the hardware aspect of our project primarily because it is a widely used hardware description language, in addition to us as the students being familiar with it's working. Both the car and the server have significant portions within their implementation that do not require code on software level in order to be realised, whether for complexity or reliability reasons. The implementation details will be discussed via detailing the ports of the entity for both parts of the project.



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity car is
  Port (
  clk : in std_logic;
  Drive:in std_logic ;
  stop :in std_logic ;
  SendInitializeRequest : out std_logic ;
  car_ID : out integer;
  server_ID:in integer;
  source: out integer range 0 to 4;
  target : out integer range 0 to 4;
  positionX:out integer;
  positionY:out integer
    );
end car;
```

Fig. 10. Entity code of Car

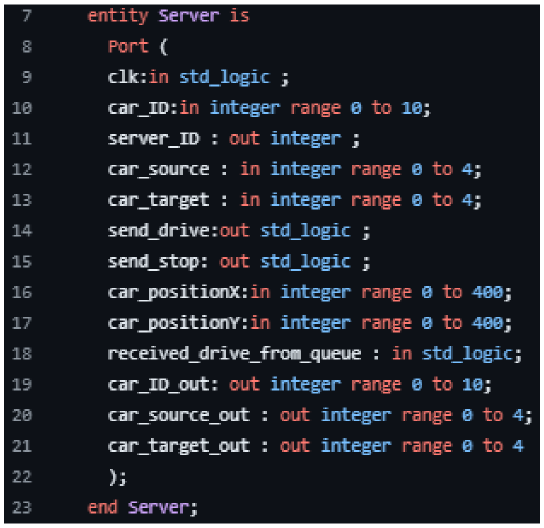
Separate entities were created for the car and the server, just by defining their inputs and outputs we are able to understand the cross communication that was done between the two entities.

Starting with the car, shown in Fig. 10. The behaviour of each of the ports is described as the following:

- **Clk**: Global clock to handle synchronous behaviour.
- **Drive**: Command sent from the server that orders it to drive.
- **Stop**: Command sent from the server that orders it to stop.
- **SendInitializeRequest**: Car sends this to the server when it is approaching the intersection.
- **Car ID**: Unique addressing of the car, that it uses to communicate with the network.
- **Server ID**: Unique addressing of the server, it receives this information once the car is initialised.
- **Source**: Car sends out which lane it is starting from.
- **Target**: Car sends out which lane it wants to go to.
- **PositionX**: Current position of the vehicle on X coordinates.
- **PositionY**: Current position of vehicle on Y coordinates.

Next is the server, as shown in Fig. 11.

- **Clk**: Global clock to handle synchronous behaviour.
- **Car ID**: Unique address received from the car that identifies it.
- **Server ID**: Unique addressing of the server, sends this information to any car that initialises on the server.
- **Car Source**: Car tells server which lane it originates from.
- **Car Target**: Car tells server which lane it wants to go to.
- **Send Drive**: No collisions detected, server tells car that it can drive.



```
entity Server is
  Port (
  clk:in std_logic ;
  car_ID:in integer range 0 to 10;
  server_ID : out integer ;
  car_source : in integer range 0 to 4;
  car_target : in integer range 0 to 4;
  send_drive:out std_logic ;
  send_stop: out std_logic ;
  car_positionX:in integer range 0 to 400;
  car_positionY:in integer range 0 to 400;
  received_drive_from_queue : in std_logic;
  car_ID_out: out integer range 0 to 10;
  car_source_out : out integer range 0 to 4;
  car_target_out : out integer range 0 to 4
  );
end Server;
```

Fig. 11. Entity code of Server

- **Send Stop**: Server tells car it is not allowed to drive.
- **Car PositionX**: Position of car on X coordinates.
- **Car PositionY**: Position of car on Y coordinates.
- **Received Drive from Queue**: Main queue informs server that the previous drive is processed, it is available to process the next drive request.
- **Car ID Out**: Passing information from car as an output.
- **Car Source Out**: Passing information from car as an output.
- **Car Target Out**: Passing information from car as an output.

REFERENCES

[1] Kenneth Yrke Jørgensen. Uppaal, May 2020.
[2] Phillip Stevens. Freertos, Jul 2024.
[3] Arduino. Arduino® mega 2560 rev3, Jun 2024.
[4] Ieee standard vhdl language reference manual. *IEEE Std 1076-1987*, pages 1–218, 1988.