Smart Traffic System: A project done for the electronic engineering lab

1st Yigitcan Aydin
Electronic Engineering
Hochshule Hamm Lippstadt
Lippstadt, Germany
yigitcan.aydin@stud.hshl.de

2nd Syed Rafsan Ishtiaque
Electronic Engineerig
Hochschule Hamm Lippstadt
ID: 2180565
syed-rafsan.ishtiaque@stud.hshl.de

***Abstract*—Smart traffic system enables a smooth, safe and time saving system for transportation. The general traffic system depends on the decision of the algorithm that most of the times doesn't update itself with respect to the actual condition of the traffic junction. In our project we demonstrate the feasibility and construction of a smart traffic system that can overcome the lacking of the conventional system. The system was developed in RTOS and demontrated in UPPAAL modelling.**

***Index Terms*—Smart Traffic, Free RTOS, UPPAAL, VHDL**

I. Introduction

Intelligent traffic system project aims to have a reliable and safe intersection passage for the autonomous cars.

In the intersection car should be able to pass without waiting for a traffic light or stopping because of the other cars that are already in the intersection. In order to solve this problem, we are using the FIFO (First In First Out) algorithm which will allow the cars pass through the intersection according to their priorities in the FIFO queue.

In order to demonstrate the entire system, first we start with the UML diagrams that explains the overall system using the use case and the activity diagrams.

Second step is to demonstrate the system behavior using a model checking software, in our project we used UPPAAL for this aspect.

Third phase is to implement the general system behavior with the C++ code implementation.

As a next step we observe the system behavior in FreeRTOS simulation. Since we compiled and ran the C++ code on our personal computer, it does not give us the proper real-time system behavior. FreeRTOS is full filling this gap in our project.

Lastly, in VHDL part, we wanted to use FIFO as a hardware in order to get faster response in putting the cars into the queue which might be crucial point in real-time implementation of the system.

II. UML Diagrams

A. Use Case diagram for the complete system

Vehicles will be detected by the sensors while approaching to the intersection. Controller will receive the data and register the vehicle. Then it will arrange the cars based on FIFO method in a queue. When cars will exit the intersection, they will be de-registered from the database
Future development: If there is a failure in the registration/ de-registration process, it will notify. For our modl, we considered reg./ de-reg. without any failure. Also, in future development, the priority among vehicles and speed orders will be added.

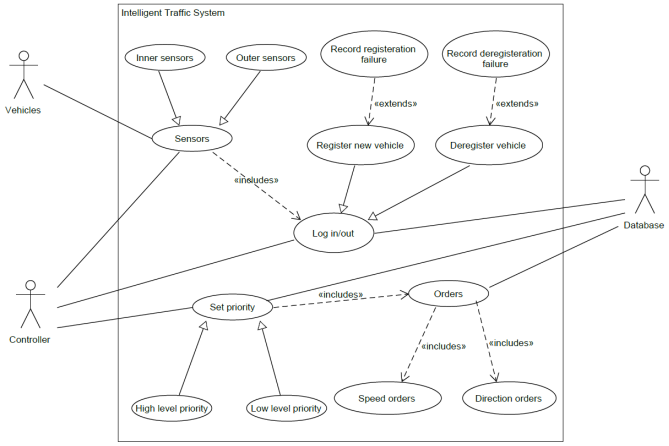

Fig. 1. Use Case diagram for the complete system with future development

B. Activity diagram for queuing the cars

Cars approaching from four directions (north, east, south, west) will be detected by sensors and send the data to controller. Controller will assign a global id for the car, and push it to the local queue with another local id based on the desired outward direction of the car. When the car will exit the intersection, a further sensor will detect it and send the data to controller.

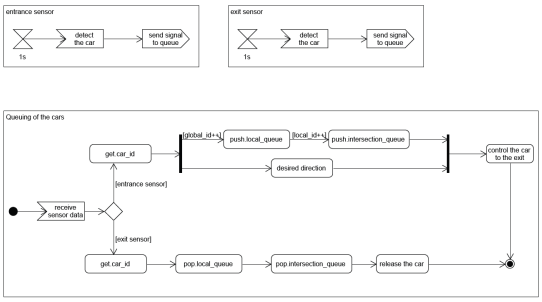

Fig. 2. Activity diagram to queue cars

III. UPPAAL Simulation

For our smart traffic system project, we developed a model to demonstrate the UPPAAL modelling of the scenario. The UPPAAL software version 4.1.26-1 (rev. 7BCF30B7363A9518), 02/2022 was used.

We considered Four cars coming from four different directions. Yes, for real life scenario it can not be always like this. For our case, we took one car from each (east, west, north, south) directions.
Here, the car from North is labelled as Car(0), West as car(1), East as Car(2) and South as Car(3).

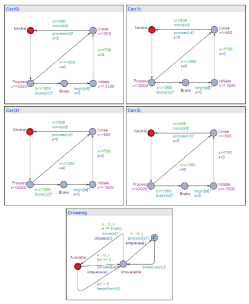

Fig. 3. UPPAAL MODELLING [1] [2]

At the beginning when all cars are outside the intersection, we see the intersection/ crossing area is available. It begins with the cars are at the Neutral state.

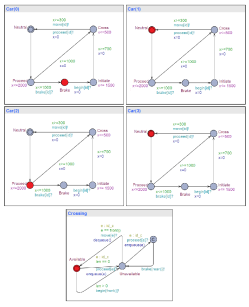

Fig. 4. UPPAAL MODELLING [1] [2]

Then (suppose one from North) first car will proceed. Here we considered a period of 2000 milliseconds before the crossing from which point till 1000 milliseconds before crossing, the car can keep proceeding, if the crossing is available. But if the crossing is unavailable the car must brake the minimum time 1500 milliseconds before the crossing. These periods/ times are approximate and can be varied in the future implementation.

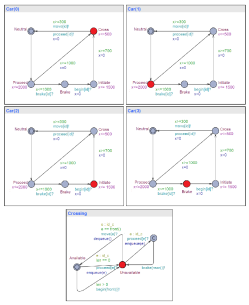

Fig. 5. UPPAAL MODELLING [1] [2]

Once the crossing is available, it will be at the begin state. So once the car is at begin state, it will be queued for the crossing. In our model we queued the cars randomly, but in future development it will be based on priority with same distance/ time gap from the crossing.

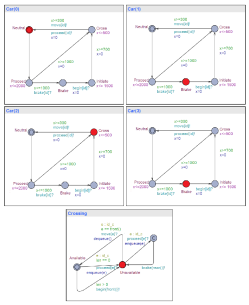

Fig. 6. UPPAAL MODELLING [1] [2]

Once the car is 300 milliseconds after the crossing, it will be in the Neutral state. So neutral state indicates the beginning at also the ending of the whole process.
Here we showed, at the beginning in Fig:3, all cars are in Neutral state. Then Fig:4, Car from North at Braking state, means still the crossing is Available as it hasn't reach the "Initiate" state. Then in Fig:5, the crossing is Unavailable, as Car(0) is in "Cross" state. Meanwhile Car(2) in Initiate state, and Car(3) in Brake state and Car(1) in Proceed state.

Then Fig:6 shows that Car(0) has successfully completed the Crossing, so it is back to Neutral state. By that time Car(2) is in Cross state. As we kept the prioritizing cars for future development, here in Fig:6, we left both Car(1) and Car(3) in the same state which is Brake state.

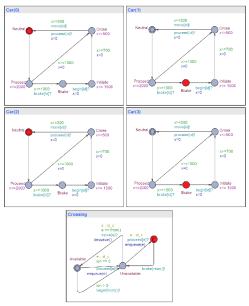

Fig. 7. UPPAAL MODELLING [1] [2]

In the last Fig:7, though there is no car at Cross stae, and only Car(1) and Car(3) in Brake state, so we have the Crossing in Committed state.
To be noted that out of four cars, only one car can be at the crossing state, but in any other states, 1 to 3 cars are possible at the same time. So when the car from North is crossing, the other cars might be in the other states, all at the same or all at different states.

IV. C++ Code Implementation

Just after creating the queues using queue library, we create the mutex for each lane on the road by using pthreadmutext. This allows us to prevent queues simultaneous access from different threads.

```
queue<int> north_lane;
queue<int> east_lane;
queue<int> south_lane;
queue<int> west_lane;
queue<int> intersection_lane;

pthread_t threadID;

pthread_mutex_t northLock;
pthread_mutex_t eastLock;
pthread_mutex_t southLock;
pthread_mutex_t westLock;
pthread_mutex_t intersectionQLock;

pthread_mutex_t global_id_Lock;
pthread_mutex_t intersectionLock;
```

We are initializing the mutex using pthreadmutexinit. After initializing it we are able to start using mutex to make sure that any other threads can reach it and change the values. This would cause our system big problems since it would destroy the behavior of the FIFO queue.

```
pthread_mutex_init(&northLock, NULL);
pthread_mutex_init(&eastLock, NULL);
pthread_mutex_init(&southLock, NULL);
pthread_mutex_init(&westLock, NULL);
pthread_mutex_init(&intersectionQLock,NULL);
pthread_mutex_init(&global_id_Lock, NULL);
pthread_mutex_init(&intersectionLock,NULL);
```

Now we lock the lanes (north, south, west, east) using pthreadmutexlock to be able to pop the car next car id into the queue. Just after pushing the id into the queue, we unlock the lane mutex and lock the intersection mutex (i.e. we send the car to the intersection with its id).

Since we have another queue in the intersection which collects the cars from north, south, west and east lanes, we do the same process as we did for the lane mutex.

```
pthread_mutex_lock(&northLock);
pthread_mutex_lock(&global_id_Lock);
localID = car_id++;
pthread_mutex_unlock(&global_id_Lock);
north_lane.push(localID);
pthread_mutex_unlock(&northLock);

sleep(5);
pthread_mutex_lock(&intersectionQLock);
intersection_lane.push(localID);
pthread_mutex_unlock(&intersectionQLock);
```

By doing the operation that has been so far explained above, we started the system, gave the cars individual IDs and let them cross the intersection depending on their IDs (or priorities) in the queues.

```
for(int i = 0; i < cars; i++)
{
  direction = getrand(1,5);
  k = getrand(1,5);
  if(k == 1)
  {
   pthread_create(&threadID,NULL,north,NULL);
  }
  else if(k == 2)
  {
   pthread_create(&threadID,NULL,east,NULL);
  }
  else if(k == 3)
  {
   pthread_create(&threadID,NULL,south,NULL);
  }
  else
  {
   pthread_create(&threadID,NULL,west,NULL);
  }
  sleep(1);
}
```

As a last mention point, we create the threads by using pthreadcreate(threadID, NULL, north, NULL). This expression allows us to create the threads for the given lane name.

V. FreeRTOS Simulation

In the FreeRTOS simulation, we demonstrate some of the conflict scenarios along with the queuing process as we did in C++ part. In order to do this we are declaring the queues with xQueueHandle to use it with xQueueCreate and xQueueSend. Former is responsible for the creation of the queues and latter is responsible for the sending mechanism of an item into the queue.

```
bool xLockLane(char Lane, char Destination);
void xUnlockLane(char Lane, char Destination);

void vNorth(void *pvParameters);
void vSouth(void *pvParameters);
void vWest(void *pvParameters);
void vEast(void *pvParameters);

xQueueHandle NorthIncomingLaneCars;
xQueueHandle SouthIncomingLaneCars;
xQueueHandle WestIncomingLaneCars;
xQueueHandle EastIncomingLaneCars;
```

Next important step is to declare the xSemaphoreHandle which will allow us to do operations with the mutex created with xSemaphoreCreateMutex. This procedure is quite similar to what we do in the C++ implementation.

```
xSemaphoreHandle WS = 0;
xSemaphoreHandle WE = 0;
xSemaphoreHandle WN = 0;
xSemaphoreHandle WW = 0;
xSemaphoreHandle SE = 0;
xSemaphoreHandle SN = 0;
xSemaphoreHandle SW = 0;
xSemaphoreHandle SS = 0;
xSemaphoreHandle EN = 0;
xSemaphoreHandle EW = 0;
xSemaphoreHandle ES = 0;
xSemaphoreHandle EE = 0;
xSemaphoreHandle NW = 0;
xSemaphoreHandle NS = 0;
xSemaphoreHandle NE = 0;
xSemaphoreHandle NN = 0;
```

In the conflict scenario, if a car coming from one of the lanes towards the intersection and has a higher ID, then we check for the possible conflict possibilities with other cars from other lanes. If there does not exist any predefined conflict possibility we let the car proceed as planned. However, if there is a possibility for an accident in the intersection, we make lower ID cars wait for the car which has higher ID first. Then we recalculate the possibilities for the remaining car in the queue.

```
if (Lane == 'W' && Destination == 'S')
{
 if (uxSemaphoreGetCount(ES)+
     uxSemaphoreGetCount(NS)==2)
 {
     xSemaphoreTake(ES,100);
     xSemaphoreTake(NS,100);
     taken = true;
 }
}
```

In the code above, we are demonstrating one of the conflict scenarios. If a car is coming from the west direction and going to the south direction, first we check for two other routes that may cause a conflict. If one of the cars is coming from the east and going to the south along with the other car is coming from the north and going to the south, then we have a conflict possibility. Therefore, we lock or take the semaphores for the routes ES and NS in order to prevent car from an accident in the intersection.

VI. VHDL Implementation

The FIFO will be generated using distributed logic or registers throughout the FPGA in a register based FIFO. This is not the same as storing a FIFO in a Block RAM. Small FIFOs (under 32 words deep) should be stored in a register-based FIFO, while bigger FIFOs should be stored in a Block RAM-based FIFO. The main goal of using the FIFO in hardware realization is to make the process faster.

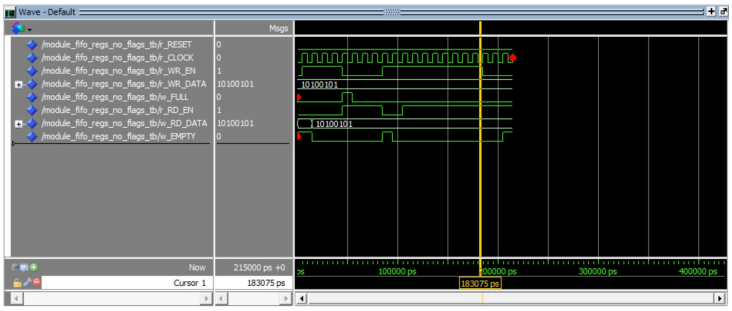

Fig. 8. Waveform generated in Modelsim for FIFO

VII. Contributions to the Project

UML diagrams: Syed Rafsan Ishtiaque, Yigitcan Aydin
UPPAAL simulation: Syed Rafsan Ishtiaque
C++ implementation: Yigitcan Aydin
FreeRTOS simulation: Yigitcan Aydin
VHDL implementation: Syed Rafsan Ishtiaque

References

[1] M. D. Wang Yi, Paul Pettersson, "Automatic verification of real-time communicating systems by const'raint-solving," 1995. [Online]. Available: https://link.springer.com/chapter/10.1007/978-0-387-34878-0$_1$8

[2] A. D. Gerd Behrmann and K. G. Larsen, "A tutorial on uppaal 4.0," 2006. [Online]. Available: https://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf