Team Wien

Shehroz Bashir Malik, Christian Stratmann, Nikolaos Karapoulatidis, Üsame Gönül
Hochschule Hamm-Lippstadt
Lippstadt, Germany
Bachelors of Engineering in Electronic Engineering

Abstract—This paper aims to illustrate the methodology and development process of the Traffic Management System. TMS is a scheduling system that manages the passage of autonomous vehicles on four different directions at a crossroad. The two main goals of the TMS is to: avoid collision at any cost, and maintain the delay at minimum. TMS system assigns a priority point to every vehicle that enters the crossroads, and schedules them accordingly. The priorities are dynamically modified depending on the attributes of the cars, like: speed, time spent in the crossroad, and direction.

1 Introduction

Traffic Management System will be explored in detail throughtout this paper. It will also be implemented. In the first section, the system is modelled and graphically illustrated using SYSML. Followed by that, the system is model checked in UPPAL. The succeeding sections detail our implementation of the Traffic Management System in FreeRTOS as well as VHDL.

2 Diagrams

2.1 Requirements Diagram

The Requirements Diagram will be found in the appendix. The Traffic Management System is refined by the following sub requirements:

1) Hard Real Time System
2) Avoid Collision
3) Accuracy
4) Reliability
5) Efficient Management
6) Modularity

The Efficient Management is further refined by the Priority Points Requirement.
Reliability is refined by Communication which in turn is refined by Attributes. Attributes contains the following: Time, Speed, Direction and Lane.

2.2 Activity Flow Diagram

The Activity Flow Diagram starts off when the System is activated. The system reads the quantities of the cars. It immediately reads the time when cars are arriving as well as their directions. The points are given based on the quantity, direction and the time spent waiting. All these 3 individual points are taken by the system and one individual point is calculated. The system then compares the points and assigns priority based on who has the highest point. The cars are allowed to pass and the system loops.

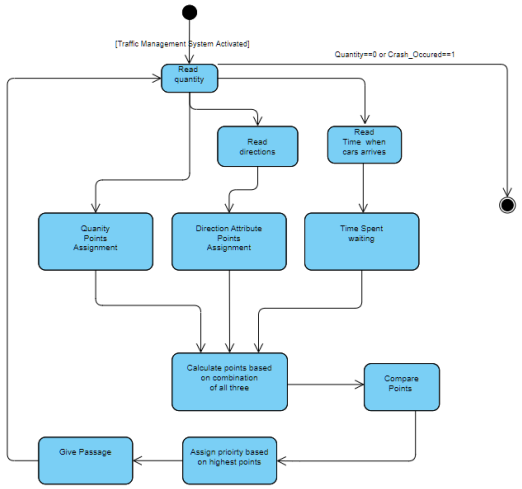


Fig. 1. Activity Diagram

2.3 Use Case Diagram

We have three individual actors for the Traffic Management System namely:

1) Cars
2) Data Collection Agencies
3) Emergency Services

The system itself captures the attributes and calculates their priorities. Cars enable this by primarily providing their attributes and travelling as per their priority.
Data Collection Agencies can record the captured attributes for their own internal statistics.
The Emergency Services have direct access to the system's manual override. This allows any emergency vehicle to gain the highest priority system and ensure they get to their destination as fast as possible.

2.4 Sequence Diagram

The cars announce their arrival to the system. The system communicates with them to acquire their attributes. The cars

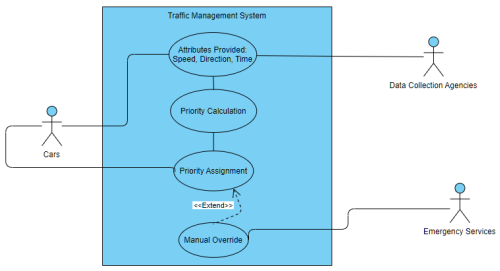


Fig. 2. Use Case Diagram

reply by sending their attributes. These are then processed for a certain amount of time. Similarly, the priorities are assigned and the commands are sent. The cars respond by updating their attributes.

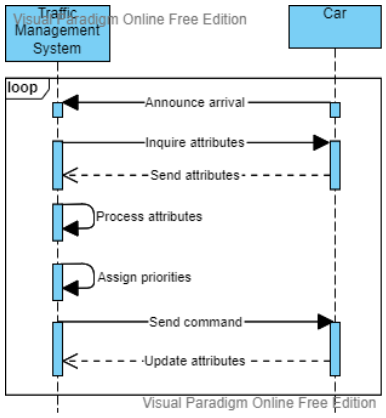


Fig. 3. Sequence Diagram

2.5 State Machine Diagram

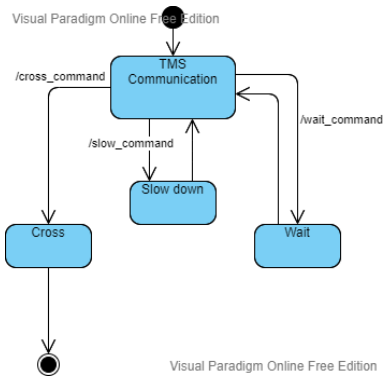


Fig. 4. State Machine Diagram

The system is initially in a state of communication. This is where it is talking directly to the cars. If it initiates the Cross Command, the state changes to Cross and the car exits. For the Slow down state, the system initiates the slow down command. After the car has slowed down, the system is allowed to return to its communication state. Similarly, for the Wait state, the system initiates the wait command. After the car has waited, the system is allowed to return to its communication state.

3 UPPAAL Implementation

The model was first implemented on UPPAAL in order to have an abstract implementation and to validate the design. UPPAAL model is based on the state machine diagram and the sequence diagram, and consists of templates for both the vehicles and for the Traffic Management System itself. The model is designed to allow communication between both parties.

In the UPPAAL implementation the Traffic Management System is made of five states, of which are: Idle, Communication, Cross, Slow Down, and Wait. The vehicle model, on the other hand, consists of four states: Idle, Announce, Wait, and Drive. The simulation starts with both models at state Idle.

As soon as a vehicle arrives to the crossroad, the vehicle model changes its state to Announce, and directly switches to Wait state, meanwhile also sending ”com” (communicate) signal to the TMS. Traffic Management System, in turn, switches to Communication state to evaluate the situation. Here, the TMS decides the vehicles next move. If, for example, the crossroad is not available, the Traffic Management System would send ”wait” command to the vehicle, after which the vehicle will stop at the crossroad and await new commands from the TMS. The command ”slow” works fundamentally similar. Instead of completely stopping, the vehicle would decrease its speed before crossing. ”cross” command, in comparison to the previous two, would return both models back into Idle state. After the ”cross” command is received by the vehicle, the state would change to Drive. As soon as the vehicle leaves the crossroads, it will send ”completed” message to the TMS, and both would finally end in the Idle state.

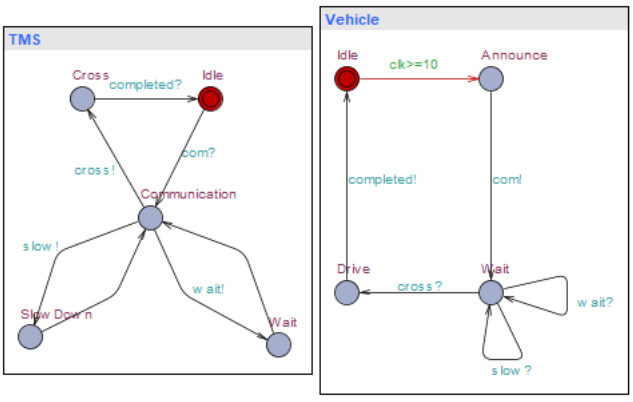


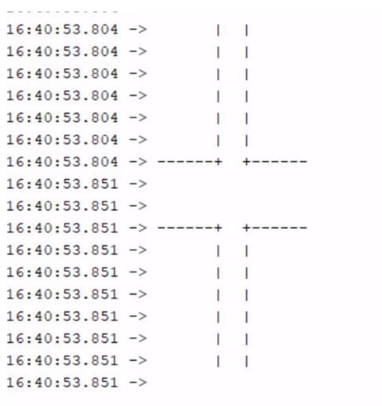
Fig. 5. UPPAAL implementation of the Traffic Management System

4 FreeRTOS Implementation

The Traffic Management System was implemented on an ESP32. The ESP32 was chosen as it had native support for FreeRTOS and shipped with a skinned version of it. The upside here is that the skinned version is quite close to the

vanilla version of FreeRTOS. We used the standard Arduino IDE with ESP32 libraries. The Code is available on our Github Repository [1]. It starts off by printing a 16x16 Map. This is updated every 500 milliseconds

.



```
16:40:53.804 ->       |  |
16:40:53.804 ->       |  |
16:40:53.804 ->       |  |
16:40:53.804 ->       |  |
16:40:53.804 ->       |  |
16:40:53.804 ->       |  |
16:40:53.804 -> ------+  +------
16:40:53.851 ->
16:40:53.851 ->
16:40:53.851 -> ------+  +------
16:40:53.851 ->       |  |
16:40:53.851 ->       |  |
16:40:53.851 ->       |  |
16:40:53.851 ->       |  |
16:40:53.851 ->       |  |
16:40:53.851 ->       |  |
16:40:53.851 ->
```

Fig. 6. FreeRTOS Map

A local car array is initiated with the following values:

1) Index(serial number)
2) Arrival time
3) From (NESW/0-3)
4) Direction(straight 0, right 1, left 2)
5) Position X
6) Position Y
7) Speed
8) Command

This local car array will eventually be synchronized with the global car array. 'From' value essentially means where is the car originally coming from - this is generated randomly. This has an additional layer of complexity where it needs to check if the spot it wants to be generated in is actually free. In the case that there is a car already there, it will search for another spot.
Direction is the path the car will be taking after generation, this is also generated randomly.
Certain commands have been abstracted to implement control. This allows the Traffic Management System to control the cars. The identifier '0' means the car is allowed to cross. '1' means the car needs to slow down and will eventually stop at checkpoint '1'. Lastly, '0' means the car has to stop and will not be allowed to drive.
Three Checkpoints have been implemented that are used to determine targets. These are

1) Before Cross Section
2) After Cross Section
3) End of Map

For the Before Cross Section, when the command is '1'; one of the axis will be modified until it arrives at the checkpoint. This in turn will change the command to '2'.
In the other case when the command is '0' and the car has arrived at Checkpoint '1' then we proceed with the next target and head to the next checkpoint. When a car at checkpoint 1 has to wait and another car comes from behind, it will wait behind that particular car.

After the Cross Section, one axis will be modified until it satisfies one of the Checkpoint's cross section. Then it will change direction. After arrival it will signal the Traffic Management System that it has finished by updating a bool array with a true.
At the End of Map, it has reached its destination and will remove itself from the map. It will also set its arrival time to '-1'. The '-1' identifier means that it is nonexistent and the Traffic Management System does not need to care about this car anymore.
Every car that has no priority points and is existent (with an identifier that is not -1) will get initial priority points for Direction, Speed and Origin. Every second, every car gets free priority points. Afterwards the queue is evaluated. It sends the first go signals and blocks all illegal directions. Once a car has finished its crossing, it removes its illegal states.

5 VHDL Implementation

We thought it is beneficial to implement the most time critical part of our scheduling algorithm into VHDL. The reason for this is that an FPGA, programmed especially for taking over the function of that part, can be extremely fast and reliable regarding execution times. In addition the function should not be too complex in order to keep the programming afford low and the flexibility high regarding adjustments of the algorithm.
We decided on implementing our "blocking function" into VHDL. This "blocking function" is also part of our FreeRTOS code since it is crucial for the algorithm.
The entity of our VHDL code consists of two input ports and four output ports. The input "car" indicates in which direction the next scheduled car wants to drive. Since we have three possible directions (straight, left, right) we use standard logic vector (2 down to 0) in order to have three different inputs for "car".
The second input shows the initial position of the car before crossing the cross-section. A standard logic vector (1 down to 0) is used. More precisely we defined "00" as North, "01" East, "10" South and "11" West.

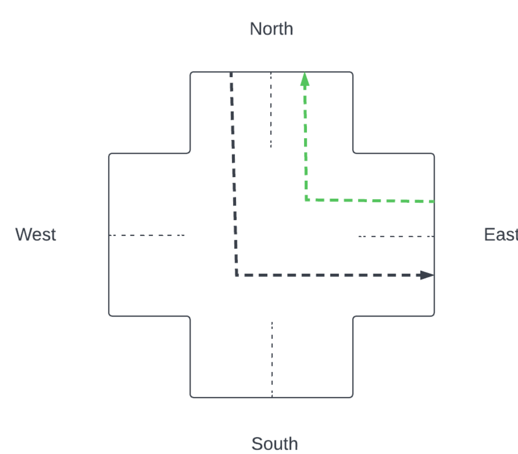


Fig. 7. Cross Section

The output of the "blocking function" is the information

on which cars are not allowed to drive while the current scheduled (e.g.first) car is crossing the cross section. This information includes the blocked point of compass and the corresponding blocked direction. So, we use a standard logic vector (2 down to 1) for each point of compass (North, East, West, South). Now, every standard logic vector also contains the information straight, right and left.
For example if a car entering the cross-section from north and wants to go left only an upcoming car from east is able to cross the street when it wants to turn right in order to avoid a crash (Figure 7).

We use a process which checks the input "car" and decides, based on the information which concurrent crossings are possible, which additional car is allowed to cross the cross-section.

6 Conclusion

The Traffic Mangement System was modeled according to SYSML Modeling Paradigms. This was followed by Model Checking in UPPAL. We successfully implemented the TMS in FreeRTOS. Lastly, achieved our desired complexity in Hardware Software Codesign by translating our functionality to VHDL.

Appendix A
Design Methodology

We followed the Agile Methodology. We had weekly Agile Sprints. In the beginning of each sprint we had sections dedicated to planning, design, coding and analysis. Initially, the team gathered together, physically or online, to have a brainstorming session. We discussed ideas to solve the problem, note down the requirements of the task ahead, adjust and incorporate our ideas to solve the aforementioned problem. An analysis was carried out to determine if our work is in line with the requirements. This is an iterative process and it is quite normal for us to restart our work if we have no productive outcomes. The tasks are split evenly amongst 4 members.

References

[1] T. Wien, "Electronic engineering a team wien's github." [Online]. Available: https://github.com/shehrozbashir546/Electronic-Engineering-A

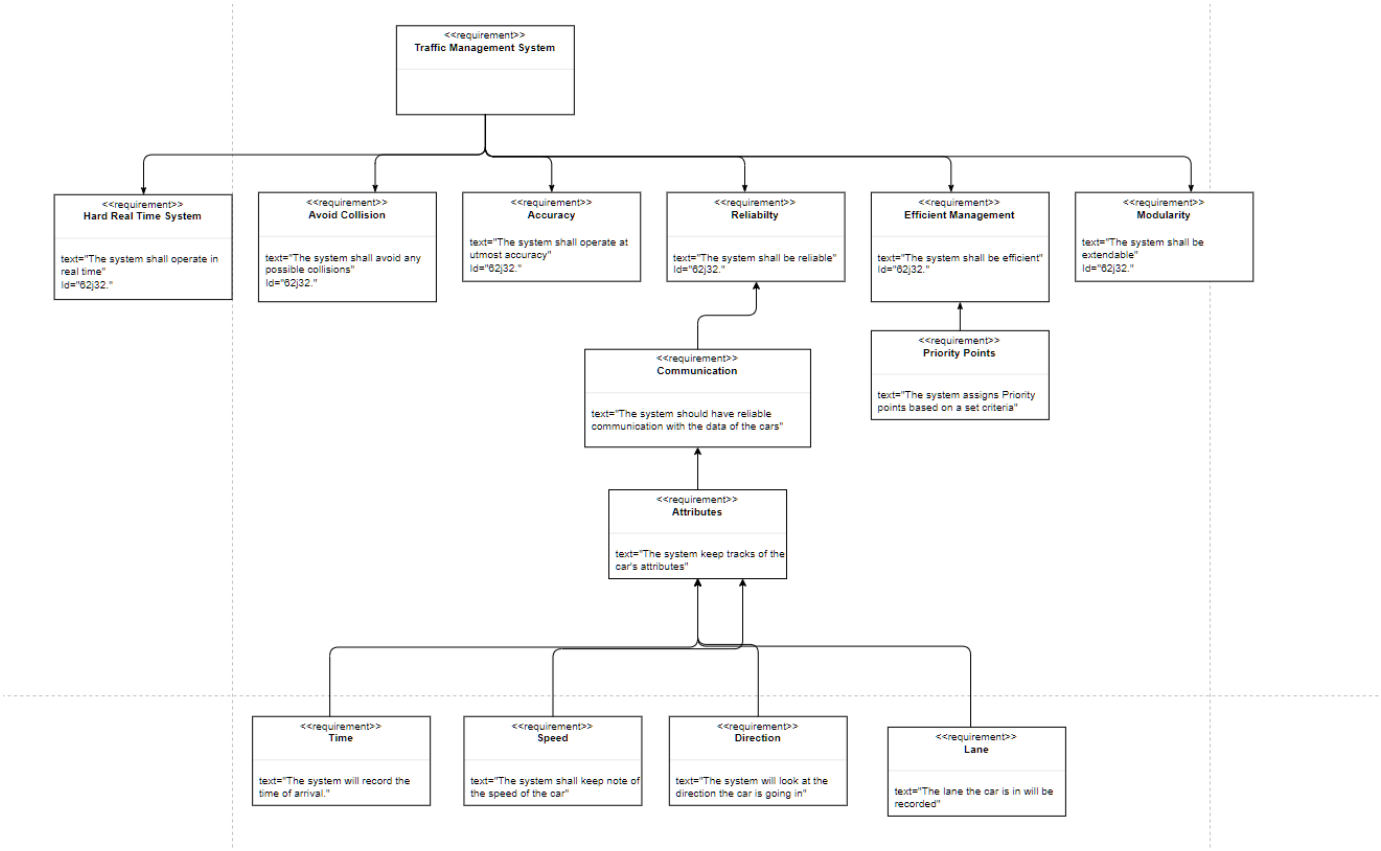


Fig. 8. Requirement Diagram