Autonomous Cross-Section Management Report

Gordan Konevski
Hochschule Hamm-Lippstadt
Lippstadt, Germany
gordan.konevski@stud.hshl.de

Aditya Kumar
Hochschule Hamm-Lippstadt
Lippstadt, Germany
aditya.kumar@stud.hshl.de

Wiktor Kochanek
Hochschule Hamm-Lippstadt
Lippstadt, Germany
wiktor.kochanek@stud.hshl.de

***Abstract*—The goal of this work will be to expand on a concept concerning a completely automated control system for a four way cross-road. First, there will be a discussion on the basis behind the problem, followed by an objective appraisal of what exactly needs to be solved. Furthermore, each of the prerequisites that have to be satisfied will be enumerated and described in the appropriate manner. In order to distinguish the hardware implementation from the software implementation, the implementation part will be divided into two distinct sections. ModelSIM and the FreeRTOS kernel, along with all of their respective libraries, will be used in both cases as the respective tools for the hardware implementation and the software implementation.**

I. Introduction

Automation of road transportation is being developed with many things in mind, but notably the improvement of: road safety, the efficiency of traffic flow, the avoidance of traffic congestion at intersections, and the time frames necessary to accomplish all of these goals. In addition to these formal considerations, various social considerations are also taken into account. One such consideration is the enhancement of the level of comfort experienced by users, especially those who are physically or mentally challenged. The applications of self-driving cars as well as interactions and interfaces with environments that are intelligent are included in the scope of the discussion of automation in this context. This last idea needs to take care of communication between vehicles as well as communication between vehicles and the surrounding infrastructure. When working within the context of infrastructure, there are many other aspects that can be taken into consideration as well. Some examples of such aspects include traffic controllers, traffic lights and pedestrian crossing sections, which, when combined, create the ideal real environment. Despite the fact that there is still a lot of work to be done when it comes to connecting autonomous driving applications to infrastructure interactions, it is generally accepted that automation in this field will become a reality sooner or later and that it will play a key role in future transportation systems. While there are already many applications of autonomous driving on the road, when it comes to connecting this to infrastructure interactions, technological advancement still has a lot of work to do.

A. Model Behaviour Explanation

Seperate parts of our model are quickly introduced and explained in the following subsections.

1) Simulation of Simplistic Version of the Model: In order to help ourselves understand the problem better, we implemented a simple version of the model using Uppaal. This simple simulation allowed us to identify pressure points of this model as well as narrow down potential solutions, which would be later implemented in the model using FreeRTOS. A detailed explanation of the implementation is provided in a later section.

2) Collision Avoidance: With safety in mind a collision avoidance part of the model was designed. The idea is for 2 cars to have implemented a hardware that communicates intended path using a 4bit signal, and by comparing signals from 2 cars, we can detect if their paths have a collision risk or not. A detailed explanation of the implementation is provided in a later section.

3) Scheduler: Following are figures 1 and 2, showing our diagrams which visualise the workings of our scheduler. A detailed explanation of the implementation is provided in a later section.

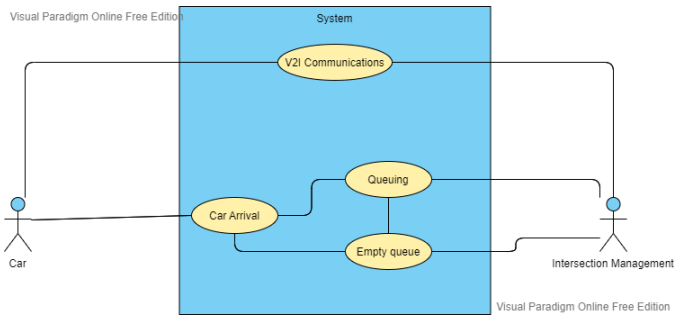


Fig. 1: Intersection Use Case Diagram

II. Behavioral Model Using Uppaal

Uppaal is an integrated tool platform for modeling, validation and verification of real-time systems. These systems are typically modeled as networks of timed automata, and Uppaal extends these networks with data types such as bounded integers, arrays, etc. Since the main intersection management algorithm will be based on a first-come-first-serve model, the Uppaal automata was modelled as such. The model is divided into these elements:

1) Car automata (Fig. 4). The car automata represents the movement and spawning of the car, through each and every stage of its path. Once the destination has been

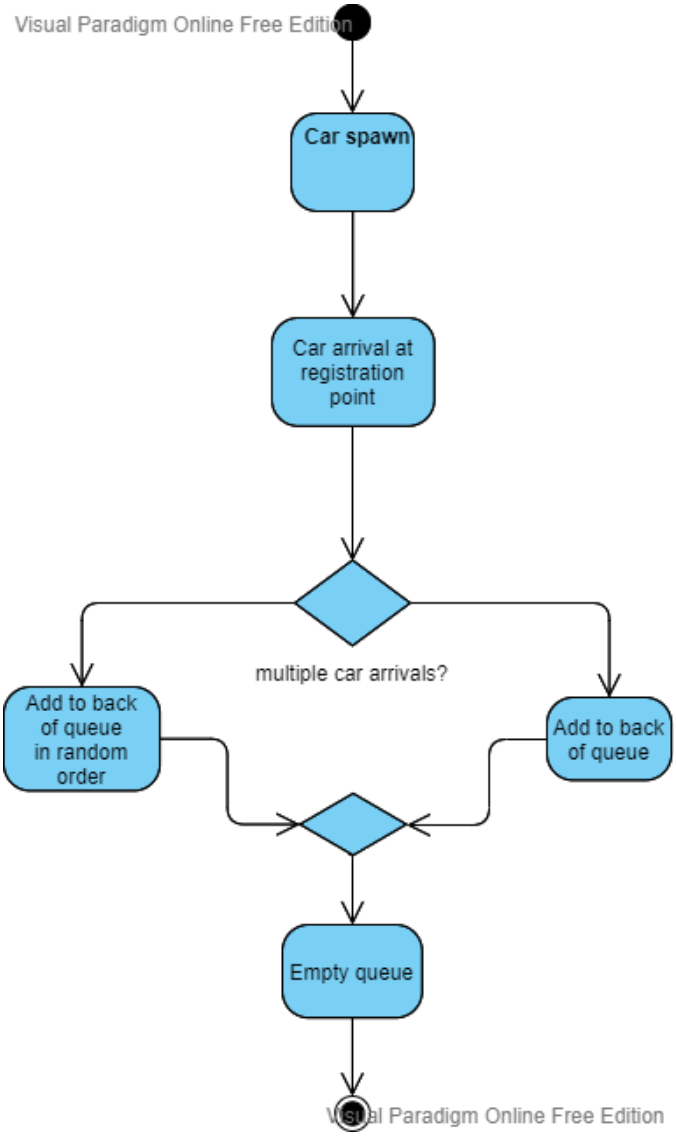


Fig. 2: Intersection Activity Diagram

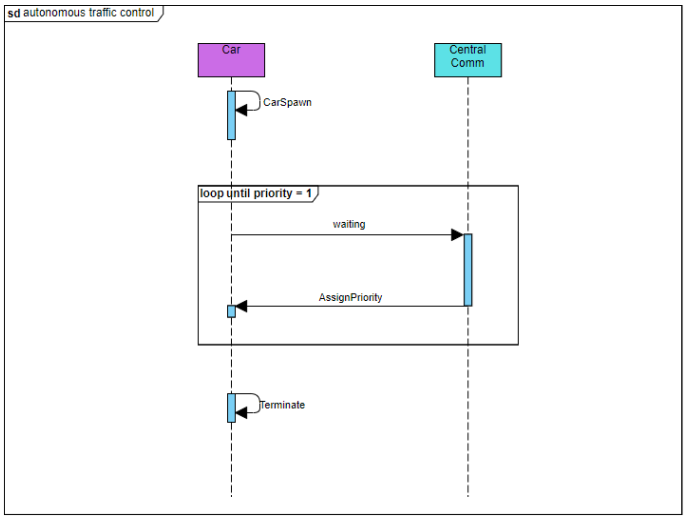


Fig. 3: Single Car Sequence Diagram

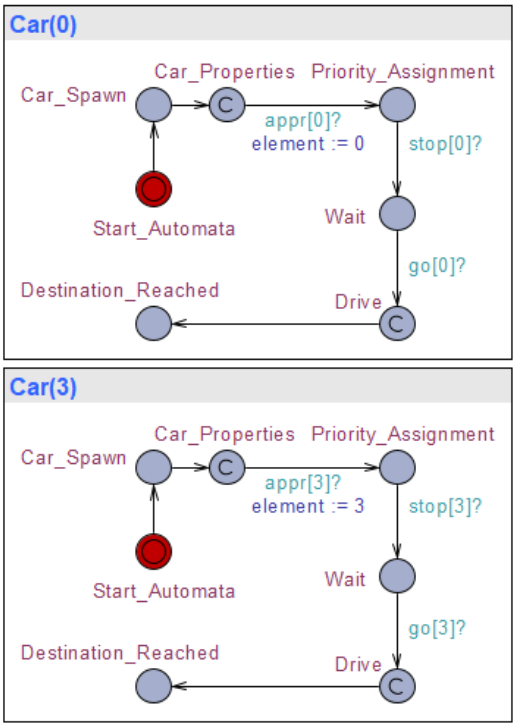


Fig. 4: Car Automata

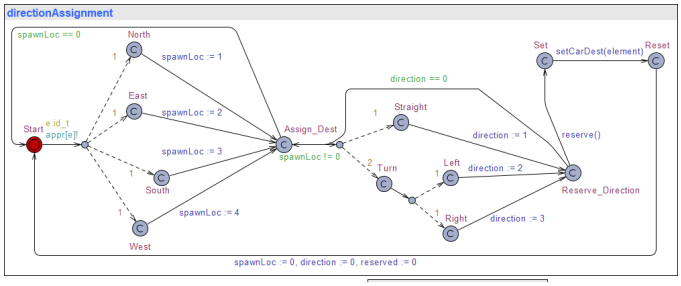


Fig. 5: Spawn and Direction Assignment Automata

reached, the automata comes to a stop. The number of cars in any given simulation can be modified in the code.

2) Direction assignment automata (Fig. 5). This automata represents the stages through which the cars get their spawn and direction properties assigned.
3) Central communication system automata (Fig. 6). The central communication system automata determines whether the vehicle is allowed to move to its desired direction and cross the intersection.

A. *Car Automata*

The car automata has a parameter of *const id_t id*, where *id_t* is a typedef variable that will be elaborated further on below. As mentioned above, the number of cars can be modified and the *id* variable enables the automata to identify each and every one. Each car begins at a starter location, from which it immediately moves to a ”Car_Spawn” location. The latter location does not execute any particular commands and is

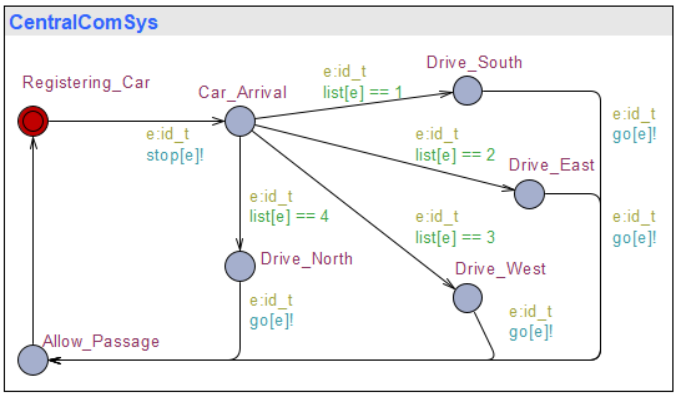


Fig. 6: Communication System Automata

mostly there for visual representation, rather than any practical application. Following that, the next location "Car_Properties" triggers the appr[id] channel which, in turn, triggers the direction assignment automata. This location is set to committed, since it is important to make sure that every car has the ability to get their properties assigned without interference from other channels. The next location, "Priority Assignment" is where the cars trigger the central communication system automata and wait for their turn to get their priority assigned. It is assumed that cars have not truly "spawned" until their priority assignment is complete. Afterwards, the cars either have to wait for their turn, or continue onward, towards their destination in the "Wait" location. The penultimate location, "Drive" is, just like the "Car_Spawn" location, there for visual representation purposes, demonstrating the vehicle moving to its target destination.

B. Direction Assignment Automata

This automata not only determines the destination of the cars, but also their point of origin. Moving from the starter location, each location, from North to South, have equal probability to be chosen. It is entirely possible for every car to be spawned from the same destination, if chance permits. Once a location has been chosen by probability, a local variable *spawnLoc* is updated. Following that, the automata continues toward the "Assign_Dest" location, which then also splits into different locations with differing probabilities, which determine whether the vehicle wants to go straight, left, or right. Again, a local variable, *direction*, is updated as a result. From the "Reserve_Direction" location a reserve() function is called, which reads the values of the local variables and determines which road the cars is going to try and reserve from the central communication system. The next location "Set" also triggers a function, which takes the data from the reserve() function and puts it in a list. The code was written as follows:

```
void reserve()
{
    if ((spawnLoc == 1 && direction == 1)
     or (spawnLoc == 2 && direction ==
     2) or (spawnLoc == 4 && direction
     == 3))
     reserved = 1;

    if ((spawnLoc == 1 && direction == 2)
     or (spawnLoc == 3 && direction ==
     3) or s(spawnLoc == 2 &&
    direction == 1))
     reserved = 2;

    if ((spawnLoc == 1 && direction == 3)
     or (spawnLoc == 3 && direction ==
     2) or (spawnLoc == 4 && direction
     == 1))
     reserved = 3;

    if ((spawnLoc == 2 && direction == 2)
     or (spawnLoc == 3 && direction ==
     1) or (spawnLoc == 4 && direction
     == 3))
         reserved = 4;

}

void setCarDest(id\_t element)
{
        list [element] = reserved;
}
```

It is worthwhile mentioning that in the (global) "Declarations" file, the following code was written:

```
const int N = 4;          // \# of cars
typedef int[0,N-1] id\_t;
```

After the automata has finished assigning values, it restarts in the starting position waiting for the next car to re-trigger it.

C. Communication System Automata

The starter location, "Registering_Car", is another visual representation of what the final algorithm aims to accomplish. This would be the point at which the car would get its id registered within the scheduler and queued for passage. However, since this is a first-come-first-serve model, the queue essentially does not concern itself with potential conflict on the intersection itself and simply allows passage on a car-by-car basis.

III. Hardware Implementation

A. Theory

Collision avoidance is a key part of considering this model for real life application as safety is a big priority. We believe the best way to implement this solution is using a hardware piece that would collect input signals from the cars relaying

information about their path direction and output a signal that would communicate if a collision between compared cares would occur or not. Using VHDL this idea is put into ralitvely simple code. A cars path would be identified using cardinal directions - North, South, East and West - denoting both the direction of where the car is coming from and where it is intending to go - for example a car approaching the intersection from the North, intending to travel to the West, would be denoted as NW. Having chosen to disregard the possibilities of 180°turns, we end up with 12 possible paths and as such we will be using a 4bit signal to define them. A table showing the definitions of those paths can be seen in figure 6.

In order to check for collisions, we had to manually create a truth table to be coded into ModelSIM in VHDL. This collision truth table can be found in figure 7 - with areas marked with X being the same path for both compared cars. The VHDL implementation would consider input signals from 2 cars and compare them using this table. If a collision is possible we output a 1, otherwise we output a 0. It can be observed that the collisions table is mirrored diagonally so to simplify the code we considered each pair only once.

N	W	S	E	
0	0	0	0	NW
0	0	0	1	NS
0	0	1	0	NE
0	0	1	1	WS
0	1	0	0	WE
0	1	0	1	WN
0	1	1	0	SE
0	1	1	1	SN
1	0	0	0	SW
1	0	0	1	EN
1	0	1	0	EW
1	0	1	1	ES
1	1	0	0	-
1	1	0	1	-
1	1	1	0	-
1	1	1	1	-

Fig. 7: Table of path definitions as 4bit signals

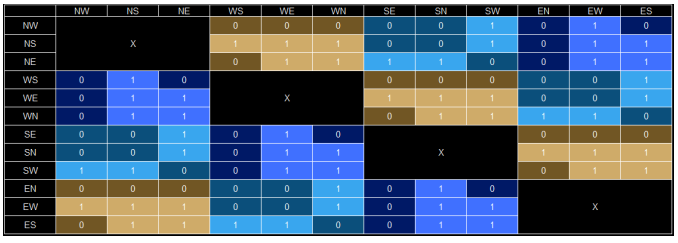


Fig. 8: Collision truth table

B. Code Implementation

Since we want the hardware to be able to compare two 4bit signals, we require 8 inputs, while for outputs, we considered a single output to be sufficient to represent whether a collision would occur or not. We define the paths of cars according to definitions from Fig. 7, but to accommodate both cars, a second set of unique variables is defined. As such we simply end up with signals denoted NW1 and NW2 identifing the same path for cars 1 and 2 respectively, ending up with 24 signal definitions in total. Following those definitions, we code logic using *when else* statements to return 1 when detecting any pair of inputs resulting in a collision and return 0 otherwise.

C. Test Bench

Considering that this implementation deals primarily with safety, when it came to our test bench we wanted to be as thorough as possible. To achieve that, our test bench took the two signals oand consideredne of them to be fixed. The second signal then ran through all of the signals defined with a direction, creating an output every time. When the running signal has had exhausted its possibilities, the fixed signal was updated by 1, and the process was repeated. Essentially, we tested every single scenario from Fig. 8, including the mirrored results and results marked as X. The only signal combinations that are not tested, are those not utilised in the code at any point - namely *1100, 1101, 1110, 1111*. However, with the code implemented in a way to return 0 when a collision pair is not detected, as well as a test bench that confirms this functionality, should any of those signals ever appear for any reason, it can be assumed that the system would return a 0. The results of our test bench, can be seen in figure 8.

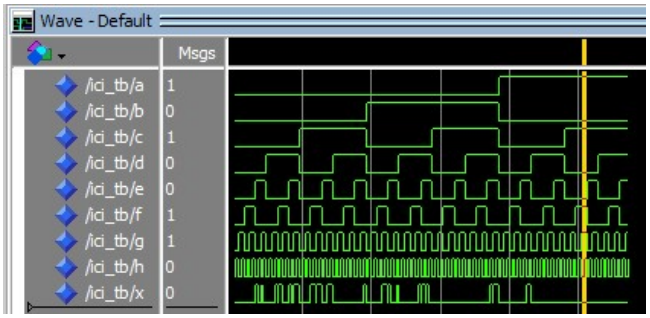


Fig. 9: Waveform of results from the test bench

IV. Software Implementation

Real-Time Operating Systems (RTOS) are widely employed in a wide variety of applications. The growth of these applications necessitates the adaptation of RTOS functionality to new requirements. In this manner, applications might be found in which the RTOS must execute not only a set of real-time tasks (RTTs), but also a set of non-real-time tasks (NRTTs). Mixed Critical Systems are systems that deal with a diverse set of tasks (that includes both RTTs and NRTTs). When an RTOS lacks support for a diverse range of tasks, it is possible

that: the timing constraints of the RTTs will not be met, the performance of the NRTTs will be inadequate, or the system resources will be wasted. There are two approaches to adapting an RTOS to support a diverse collection of jobs. The first is to change the kernel to add new scheduling policies for this type of set of tasks. The second option is to provide a user-level scheduler that overrides the operations of the kernel scheduler. The former option necessitates changes to the RTOS kernel, which are often complex from multiple perspectives: it requires a thorough understanding of the RTOS structure and can result in the loss of safety certifications, robustness, and reliability, though such changes result in a more efficient use of resources. Unlike the first approach, the second allows the designer to quickly modify the RTOS to the application's scheduling requirements. The RTOS safety certifications, robustness, and dependability remain unchanged, and should be sufficient to validate the produced application. All of that notwithstanding, our algorithm is a system of RTTs and was designed with that in mind. The software implementation of the project, ultimately, was simulated on an ESP 32 micro controller. ESP 32 board already comes with the the FreeRTOS firmware installed on it. We used an online ESP 32 simulator to simulate our results.

A. The Algorithm

The algorithm retains the first-come-first-serve nature of its Uppaal counterpart. Each car gets their spawn point and direction determined and then try to access a scheduler task in real-time. The algorithm, though, implements a pseudo-delay, meaning the time at which the cars initiate a message response with the scheduler is delayed artificially. This is done in order to simulate the time it would potentially take for the cars to reach the distance necessary in order to communicate with the scheduler. Afterward, it is a matter of queuing the priority of passage of the cars in proper order and stopping cars that might cause conflict with the queue, or in the intersection, in general.

B. The Implementation

The algorithm was set-up with four main tasks, the first three of which will be focused on:

1) void *vSpawnCarTask*(void *pvParameters)
2) void *vSchedulerTask*(void *pvParameters)
3) void *vResolverTask*(void *pvParameters)
4) static int *car_comparator*(const void* c1, const void* c2)

Just like in the Uppaal implementation, there is a section that specifically deals with the properties of the car (see Fig. 10), although this time, the implementation considers the temporal aspect of the problem. Data like the cars' direction and id remain quite similar.

The scheduler on the other hand is quite different from its Uppaal counterpart (see Fig. 11). It can be seen that the scheduler has full control of the buffer on which the cars rely on and is also responsible for menial error-handling via

```
// spawn the car
void  vSpawnCarTask(void *pvParameters) {

    BaseType_t xStatus;
    TickType_t delay = ((rc_settings_t*)pvParameters)->delay;
    int from_lane = ((rc_settings_t*)pvParameters)->from_lane;

    for ( ;; ) {
        volatile TickType_t call_time = pdMS_TO_TICKS(xTaskGetTickCount());

        Vehical_t vehical = {
            .v_id = (rand() % 10000) + 1,
            .direction =  3*from_lane + (rand() % 3),
            .poll_time = call_time,
            .scheduled = false
        };

        xStatus = xQueueSendToBack( xUnscheduledCarsQueue,(void *) &vehical, portMAX_DELAY );
        vTaskDelay(pdMS_TO_TICKS(delay));
    }
}
```

Fig. 10: FreeRTOS vSpawnCarTask

```
void  vSchedulerTask(void *pvParameters) {

    Vehical_t xReadBuffer[4];
    BaseType_t xStatus;

    for( ;; ) {
        xStatus = xQueueReceive(xUnscheduledCarsQueue, &xReadBuffer[0], portMAX_DELAY);

        int buffer_fill = 1;
        for (int i = 1; i < 4; i++) {
            vTaskDelay(pdMS_TO_TICKS(50));

            if (uxQueueMessagesWaiting(xUnscheduledCarsQueue) == 0) {
                break;
            }
            else {
                xStatus = xQueueReceive(xUnscheduledCarsQueue, &xReadBuffer[i], 0);
                buffer_fill++;
            }
        }

        if (buffer_fill == 1) {
            Serial.print("Single car received, pushed immediately...\r\n");
            xReadBuffer[0].scheduled = true;
            xStatus = xQueueSendToBack( xScheduledCarsQueue, &xReadBuffer[0], 0);
        }
        else {
            qsort(xReadBuffer, buffer_fill, sizeof(Vehical_t), car_comparator);

            for (int i = 0; i < buffer_fill; i++) {
                Serial.print("scheduled vehical\r\n");
                //Serial.print("Car scheduled\r\n" + xReadBuffer[i].v_id);
                Serial.print(xReadBuffer[i].v_id);
                Serial.println();
                xReadBuffer[i].scheduled = true;
                xStatus = xQueueSendToBack( xScheduledCarsQueue, &xReadBuffer[i], 0);
            }
            Serial.print("Many cars received, all pushed...\r\n");
        }
    }
}
```

Fig. 11: FreeRTOS vSchedulerTask

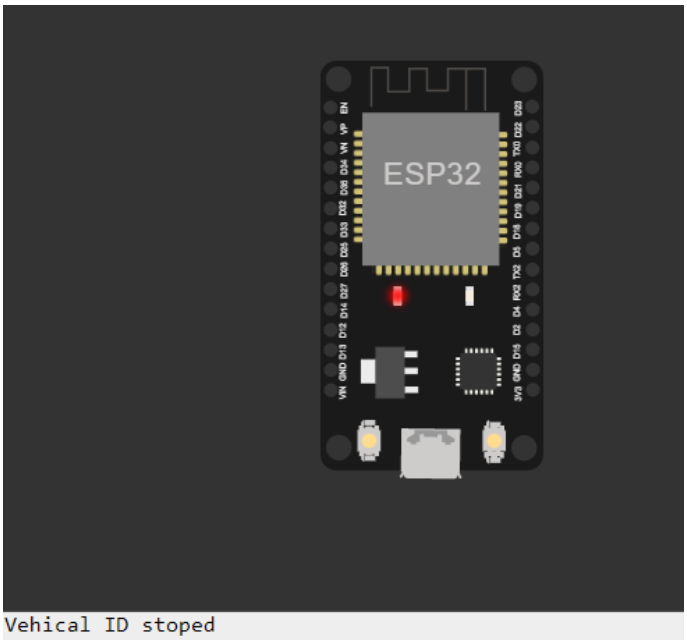
messages. Finally, there is a resolver, which is ultimately responsible for clearing the queue of cars that have reached their destination (simulated by a time ticker) and also handles potential conflicts between cars by enforcing the priority set by the scheduler.

V. Conclusion

Although not every aspect of intersection management could be taken into account, this paper demonstrates that simple solutions can be effective enough on their own. Naturally, the complexity of the algorithm demonstrated could be increased further, but that would increase the risk involved in the

```
// resolver
void vResolverTask(void *pvParameters) {
    Vehical_t xReadBuffer[2];
    BaseType_t xStatus;
    bool last_stoped = false;
    for ( ;; ) {
        if (last_stoped) {
            xReadBuffer[0] = xReadBuffer[1];
            last_stoped = false;
        }
        else {
            xStatus = xQueueReceive(xScheduledCarsQueue, &xReadBuffer[0], portMAX_DELAY);
        }
        xStatus = xQueueReceive(xScheduledCarsQueue, &xReadBuffer[1], pdMS_TO_TICKS(50));
        if (xStatus == pdFALSE) {
            Serial.print("Vehical ID let through\n");
            Serial.print( xReadBuffer[0].v_id);
            Serial.println();
        }
        else {
            const int ms_threshold = 3000;
            if (xReadBuffer[0].poll_time - xReadBuffer[1].poll_time < ms_threshold) {
                Serial.print("Vehical ID let through\n");
                Serial.print( xReadBuffer[0].v_id);
                Serial.println();
                Serial.print("Vehical ID stoped\n");
                Serial.print( xReadBuffer[1].v_id );
                Serial.println();
                last_stoped = true;
            }
            else {
                Serial.print("Vehical ID's let through \n");
                Serial.print( xReadBuffer[0].v_id);
                Serial.println();
                Serial.print( xReadBuffer[1].v_id);
                Serial.println();
                Serial.println();
            }
        }
    }
}
```

Fig. 12: FreeRTOS Serial Print




```
Vehical ID stoped
9712
scheduled vehical
2844
scheduled vehical
```

Fig. 13: FreeRTOS Serial Print

scheduling process. In a situation where most cars do not share the same standards, a simpler system might just prove more fruitful.

Declaration of Originality

We hereby confirm that we have written the accompanying paper by ourselves, without contributions from any sources other than those cited in the text and acknowledgements. This applies also to all graphics, drawings, maps and images included in the paper. The paper was not examined before, nor has it been published.

References

[1] https://uppaal.org/
[2] https://wokwi.com/