# Computer Modeling Final Project Report

Puscasu Razvan Stefan
student id: 2023-81117

## 1 Introduction

The goal of the project was to simulate the Electronics Control Unit (ECU) of a car using the framework provided by the professor. The technical details of the simulation follow the model outlined in Functionally and Temporally Correct Simulation of Cyber-Systems for Automotive Systems (Kyoung-Soo We, Seunggon Kim, Wonseok Lee and Chang-Gun Lee). Our task was to design the Lane Keeping and Cruise Control of the car to follow the given requirements of: speed cannot exceed 80km/h, no deadline misses and also include Sensing, Brake, Power and Local.
The project setup involved two laptops. One ran the simulation software which follows the model mentioned above and another laptop takes the data provided by the simulation and uses it to control a car inside of TORCS (The Open Racing Car Simulator). The second laptop also provided feedback back to the first one (for example the position, for lane keeping and speed for cruise control). Finally, the two laptops communicated via an ethernet cable.

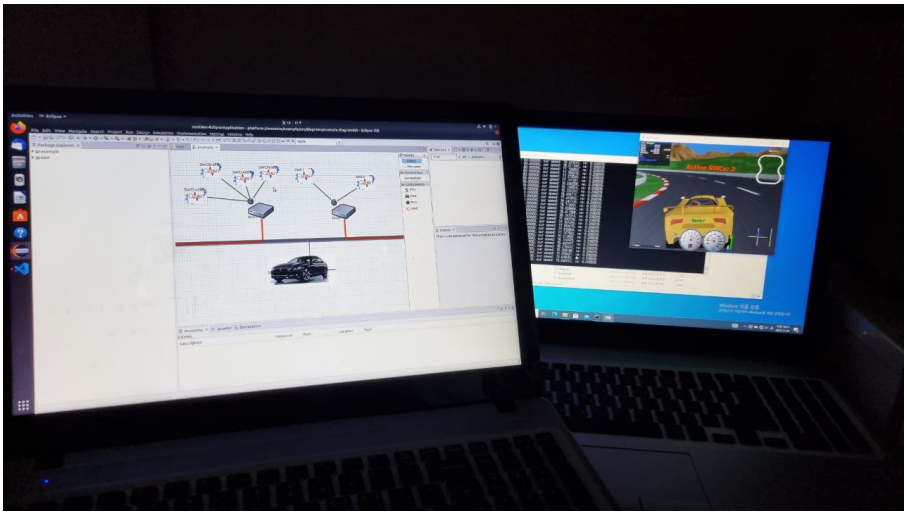

Figure 1: The two laptops used for the simulation

## 2 Making the car as fast as possible

The first part of the project involved making the car as fast as possible on a track. The restrictions we had to follow were the following: cruise control cannot be set above 80km/h, there can be no deadline misses and we had 6 SWC components in total, for lane keeping, cruise control, sensing, brake, power and local.

Initially, I understood that the task was to make the car as fast as possible with no speed limit and so I managed to obtain a time of 53s with a speed of 123km/h and 2 ECUs. After taking into account the maximum speed limit of 80km/h I obtained a time between 1m16s and 1m20s. I kept the setup with 2 ECUs for the sake of simplicity. When I tried using a single ECU the car wasn't able to stay on the road. In the picture bellow is a table of the timings that I used:

## 3 Logging the data

Logging the data is a fundamental part of debugging the functionality of the simulation. As such we had to write logs of the events happening inside the ECU, when data is being read and when data is being written. We also had to write logs for the real cyber system, this is for the cases when logging the data reads and writes is not enough, and we need to see the schedule of the real system.

| | LK | CC | SENSING | BRAKE | POWER | LOCAL |
|---|---|---|---|---|---|---|
| Period(Period, Deadline) | 200 | 200 | 500 | 500 | 500 | 500 |
| Execution time(WCET,BCET) | 80 | 80 | 100 | 100 | 100 | 100 |
| Read Constraint | Y | Y | N | N | N | N |
| Write Constraint | Y | Y | N | N | N | N |

Figure 2: Table of timings used

### 3.1 Case #1

In this first case I initially tried the approach outlined in the project guidelines. This consisted of implementing a separate function for logging that gets called inside of "Job.cpp". This task proved to be very difficult as using the shared pointer of the running job (running_job) caused the car to not move. After some investigation and looking at the other logs, I realized that this was probably caused by a deadlock, as the other logger from "Logging.cpp", which ran in a separate thread only wrote the first 300ms of logs, even though the simulation ran for close to a minute. Since deadlocks and similar problems are notoriously hard to track down, and I was unfamiliar with the codebase, I decided to drop this approach and instead do the logging directly in the "Job.cpp" file. This method provided me with satisfactory results immediately. Bellow is a photo of the code that does this logging in the "Job.cpp" file:


Figure 3: Case #1 logging logic, inside "Job.cpp"

### 3.2 Case #2

In the second case we had to write the logs for the scheduling of the real cyber system. This is necessary for when the logs from the first case look good, but the car is still not moving. That issue could be caused by deadline misses which only the scheduling of the real cyber system can catch. Keeping this motivation in mind, and the shortcomings of my initial approach from the first case for logging I decided to directly implement this inside of the "Executor.cpp" file so I wouldn't run into similar hard-to-debug issues again.

My approach consisted of creating a custom struct to store the information required of the tasks, and storing all instances of that struct in a std::vector. Using a std::vector was the easiest approach so I wouldn't need to keep track of the size of the array, which could be another potential cause of issues. There are 2 important for loops that I used, one is used by the executor to check for deadline misses and another one does the actual scheduling. Inside both for loops I record the events time, job id, and event type. Finally, after the scheduling is finished inside of a hyperperiod, I sort the std::vector and print it's contents to a file. Using this approach means that I only keep the last hyperperiod of events. On the next page are the pictures of the code that I wrote.

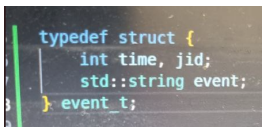


Figure 4: Logging struct

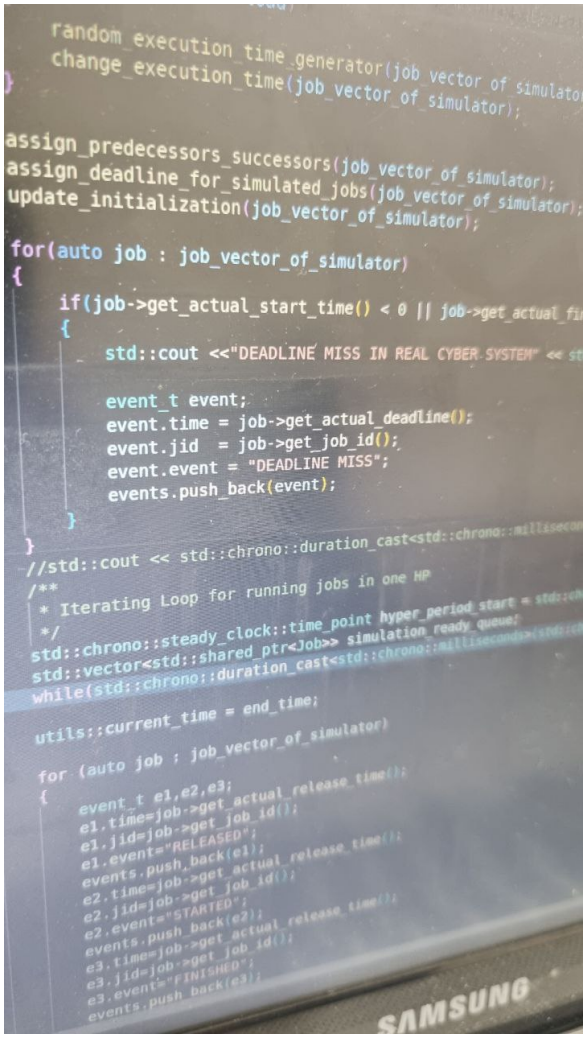


Figure 5: Getting the deadline misses

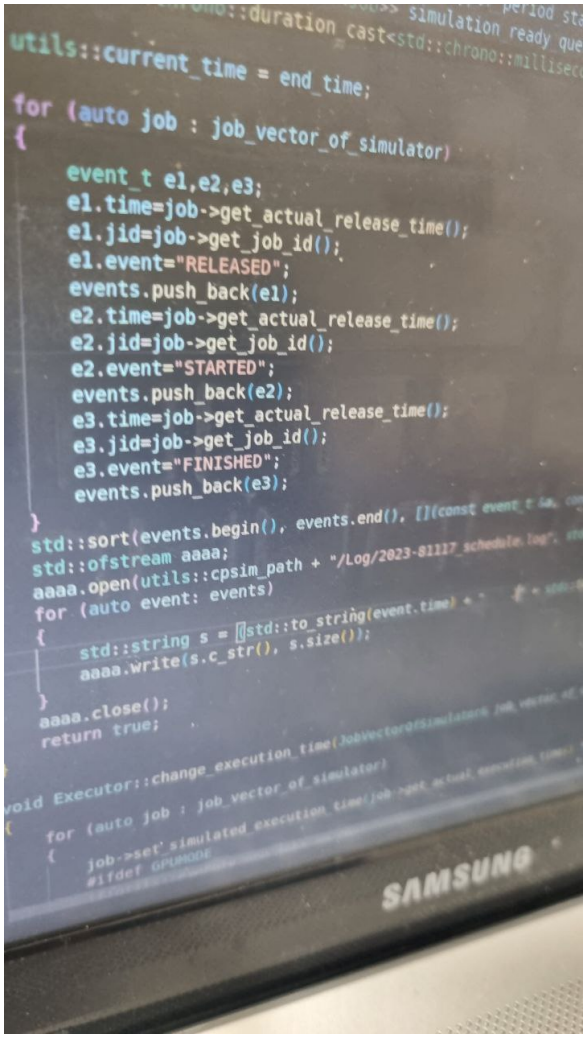


Figure 6: Logging information of regular events

# 4 Conclusions

This project was a great opportunity to learn about how real-time system can work in the context of a car's ECU. Personally, I really enjoyed having a simplified overview of how a complicated embedded system may work in reality. Being able to directly manipulate the simulation was crucial to understanding how the simulated system functions. Furthermore, I think that the logging techniques we used for this simulation will prove to be useful in the future when working on any type of large system. Before working on this project I mostly used rudimentary printf/cout statements for debugging, however that might not be suitable for a complex program like the simulation software.

On the other hand, adding new components to a large codebase that I did not fully understand was difficult. One of my biggest problems was not being able to understand what caused the unwanted behaviour when I added new functions. Furthermore, because of the multithreading used, all of the "bad" or rather untested code caused very complex and almost impossible to debug issues like deadlocks. As a consequence I resorted to using simpler methods to get the logging by modifing the functions where I wanted to do the logging instead of calling a separate function.

Finally, I want to conclude by saying that despite the difficulties it was an amazing learning experience for me

to finally work on a large codebase and write code that performs tasks correctly.