Autonomous Vehicles
Assignment V: Finite State Machines

Tommaso Bocchietti 10740309

A.Y. 2024/25

POLITECNICO
MILANO 1863

Contents

List of Figures

1 Introduction

The aim of this work is to gain insight into the modelling of systems via Finite State Machines (FSMs) and to understand the principles behind their implementation in the context of autonomous vehicles.
At first, a brief overview of the concepts of FSMs comprehensive of simple examples is provided. Then, the focus shifts to the modelling of a simple application, namely a parking gate system, which is designed using FSMs.

Tools As for the tools used, `Simulink` and in particular `Simulink/Stateflows`, is employed as the main tool for implementing the FSM and simulating the system. On the other hand, `MATLAB` is simply used to plot the results of the simulation.

2 Finite State Machines

Finite state machines (FSMs) are a powerful method of modelling systems whose output depends on the entire history of their inputs (as opposed to memoryless systems, where the output depends on the current input only). FSMs are widely used in various fields, including computer science, control systems, and digital circuit design. They can be classified into two main classes: deterministic finite state machines (DFSMs) and non-deterministic finite state machines (NDFSMs).
Mathematically, a finite state machine is defined as a 6-tuple $(S, I, O, \delta, \lambda, s_0)$, where:

- S is a finite set of states.
- I is a finite set of input symbols (input alphabet).
- O is a finite set of output symbols (output alphabet).
- $\delta : S \times I \to S$ is the state transition function, which maps a state and an input symbol to the next state.
- $\lambda : S \times I \to O$ is the output function, which maps a state and an input symbol to an output symbol.
- $s_0 \in S$ is the initial state.

The FSM processes a sequence of input symbols, transitioning between states according to the state transition function δ and producing output symbols according to the output function λ. The behavior of an FSM can be represented using state transition diagrams or state transition tables.

2.1 Turnstile FSM

As an example, we consider a turnstile that allows entry when a coin is inserted and locks when a person pushes through. The FSM for this turnstile can be represented by the state transition diagram in Figure 1 [1] and the state transition table in Table 1 [1].

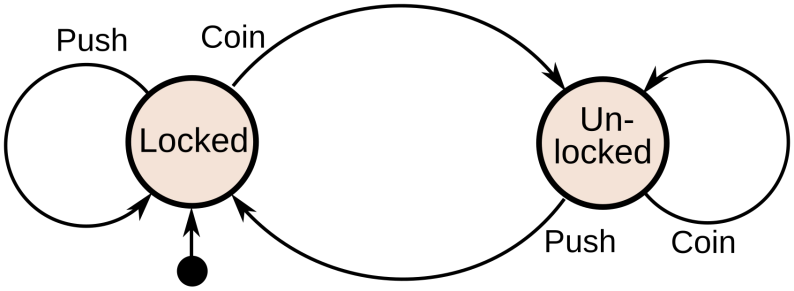

Figure 1: Turnstile FSM State Transition Diagram

Current State	Input	Next State	Output
Locked	Coin	Unlocked	Unlocks the turnstile so that the customer can push through
	Push	Locked	None
Unlocked	Coin	Unlocked	None
	Push	Locked	When the customer has pushed through, locks the turnstile

Table 1: Turnstile FSM Transition Table

The FSM starts in the *Locked* state. When a coin is inserted, it transitions to the *Unlocked* state and allows the customer to push through. If the customer pushes through while in the *Unlocked* state, the FSM transitions back to the *Locked* state. If a coin is inserted while in the *Unlocked* state, the FSM remains in the same state.

3 Parking Gate System

As already mentioned in the previous sections, the focus of this assignment is to model a parking gate system using finite state machines (FSMs).

3.1 Requests

The requests that the parking gate system has to fulfil are the following:

- Analyze the provided *Vehicle FSM* and understand how it works;
- Define the *raise* and *lower* functions of the gate respecting $w_{max} = |4|[^\circ/s]$ and $\dot{w}_{max} = |1|[^\circ/s^2]$;
- Define the FSM to model the parking gate control system;
- Integrate the *Vehicle FSM* and the parking gate control system FSM;
- Provide results of the simulation of the integrated system;

Notice that despite the suggestion from the professor to start from the already provided *Vechicle FSM*, I decided to start from scratch in order to develop a deepened understanding of the FSMs, modelling also events, slightly more complex logics and composition / decomposition patterns.

3.2 System Overview

The parking gate system is composed of a single Finite State Machine (FSM) that is responsible for modelling at the same time the vehicle and the parking gate system.

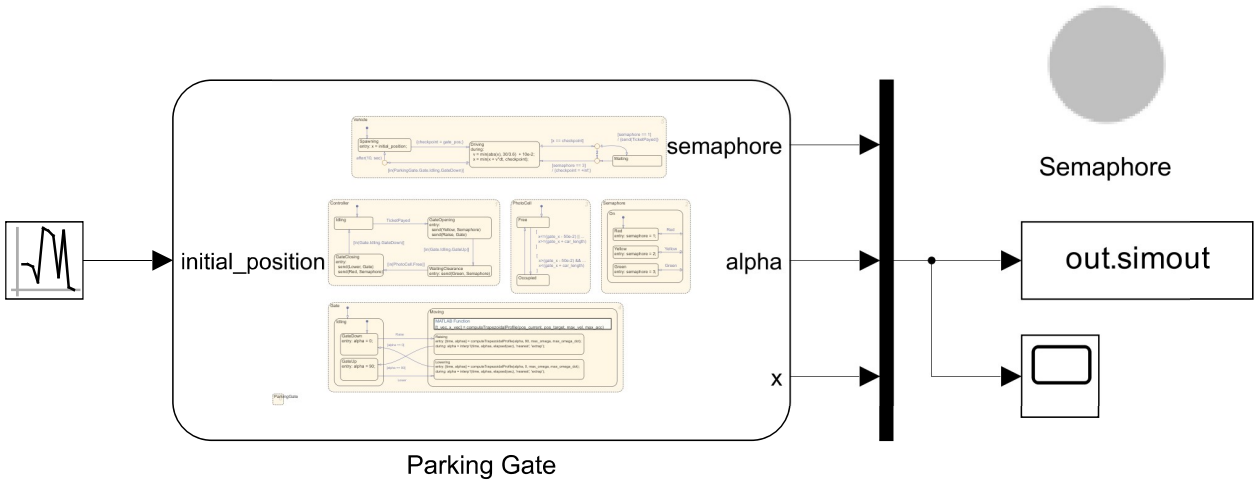

Figure 2: Parking Gate System Overview

Under the hood of the unique FSM, the system is composed of other two parallel FSMs, with precise and well-defined roles and responsibilities. The communication between the two FSMs is done through events and state transitions, which are used to inform each other about the current state of the system and to trigger the appropriate actions.

- The *Vehicle FSM* share the *vehicle_position (x)* with the *Parking Gate FSM* to inform it about the vehicle's position in the parking lot;
- The *Parking Gate FSM* share the *semaphore_state* with the *Vehicle FSM* to inform it about the state of the gate (closed, opening, opened);
- The *Vehicle FSM* also sends the event about the *TicketPayed* to the *Parking Gate FSM* to inform it that the vehicle has paid for the ticket and is ready to leave the parking lot;

Figure 2 shows the top level of the parking gate system, while Figure 3 shows the parking gate system FSM.

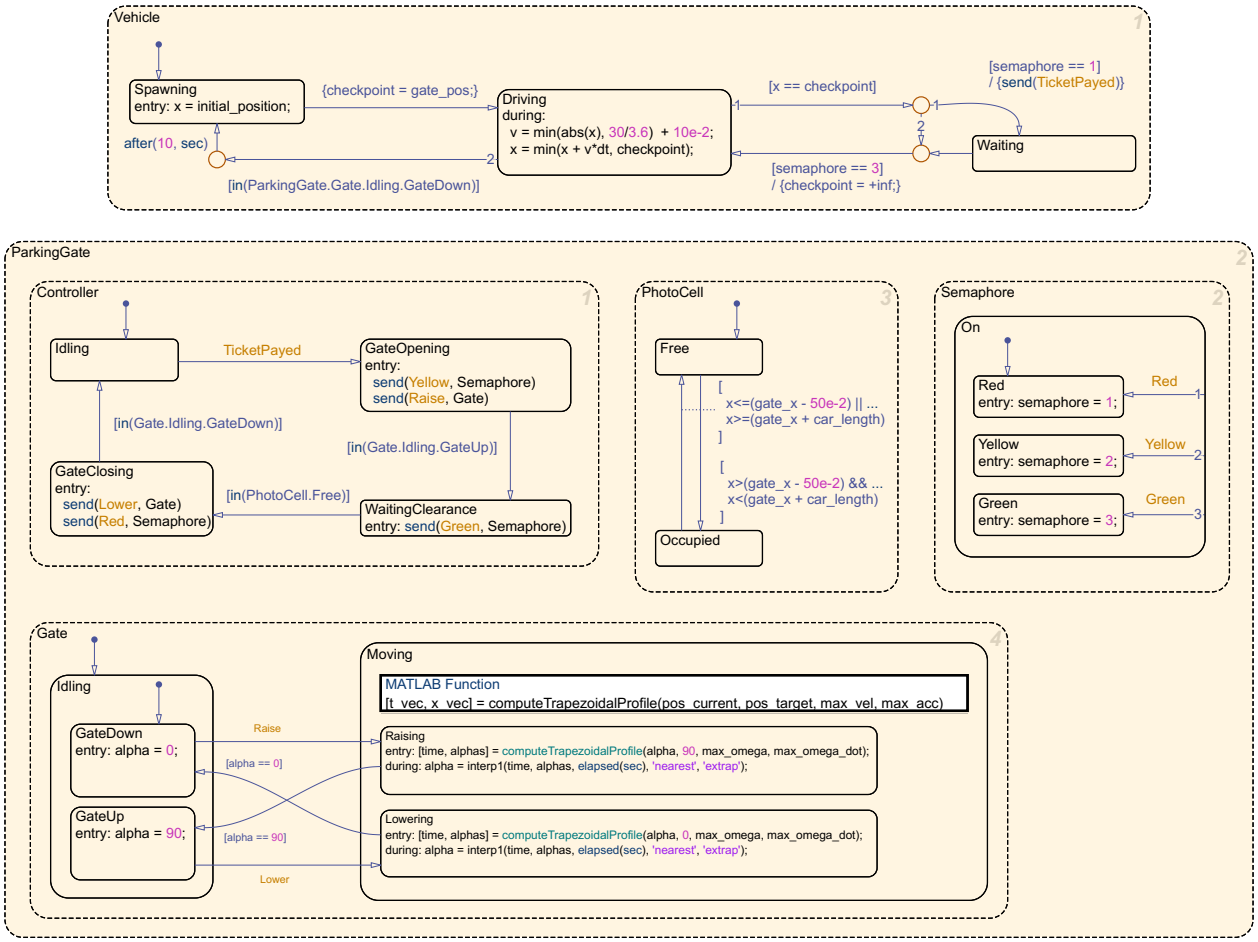

Figure 3: Parking Gate System FSM

3.2.1 Vehicle FSM

The vehicle FSM is responsible for simulating the vehicle's behavior in the parking lot. It has two main states: *Driving* and *Waiting*. At first, the vehicle is initialized and given a random position ahead of the gate. A checkpoint is also set so that it will stop at the gate before proceeding to the next state. Once at the gate, the vehicle looks for the semaphore state: if the semaphore is green, it directly returns to the *Driving* state and proceeds ahead to the next checkpoint which is now set to $+\infty$; instead, if the semaphore is red, it simulates the payment of the ticket by sending the event *TicketPayed* to the parking gate FSM and goes into the *Waiting* state, waiting for the semaphore to turn green. The vehicle proceed until the gate is closed again, when the external environment resets the vehicle to a new random position ahead of the gate to start the process again. Figure 4 shows the vehicle FSM.

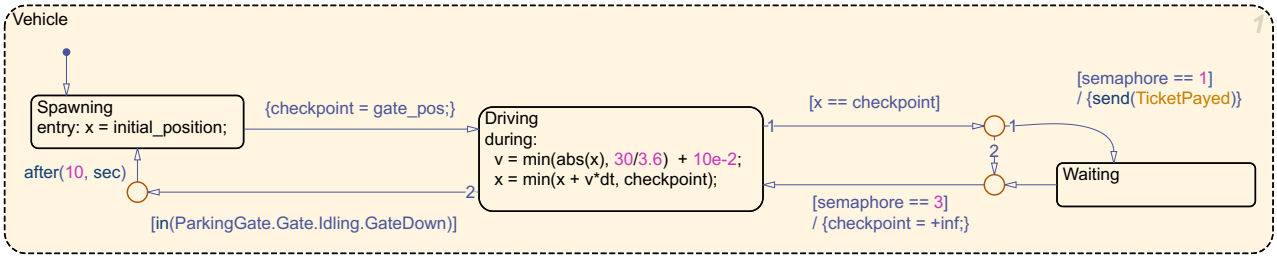

Figure 4: Vehicle FSM

Notice that the vehicle FSM has been modelled adopting the default *OR* logic, meaning that the vehicle can assume only one state at a time (i.e. *Driving* or *Waiting*, or *Spawning* when the vehicle is initialized).

3.2.2 Parking Gate FSM

The parking gate FSM is responsible for controlling at first the gate's opening and closing, and then the semaphore state. It has four main blocks: *Controller*, *Gate*, *Semaphore* and *PhotoCell*.

Controller The entry point of the parking gate FSM is the *Controller* block, which is responsible for managing the overall state of the system and coordinating the other blocks. It has been designed to be event-driven, meaning that communicates with the subcomponents of the system by sending and receiving events. It has four main states: *Idling*, *GateOpening*, *WaitingClearance* and *GateClosing*. At first, the *Controller* block is in the *Idling* state, waiting for the vehicle to arrive and receive the *TicketPayed* event. Once the ticket is paid, the *Controller* block moves to the *GateOpening* state, where it sends the *Raise* event to the *Gate* block and the *Yellow* event to the *Semaphore* block. Once the gate is fully opened, the semaphore turns green and the *Controller* block moves to the *WaitingClearance* state, where it waits for the vehicle to pass through the gate. Notice that the system determines the vehicle's position by using the *PhotoCell* block, which is responsible for detecting the vehicle's presence in front of the gate. Once the vehicle has passed through the gate, the *Controller* block moves to the *GateClosing* state, where it sends the *Lower* event to the *Gate* block and the *Red* event to the *Semaphore* block. The *Controller* block then moves back to the *Idling* state, waiting for the next vehicle to arrive.
Figure 5 shows the *Controller* block of the parking gate FSM.

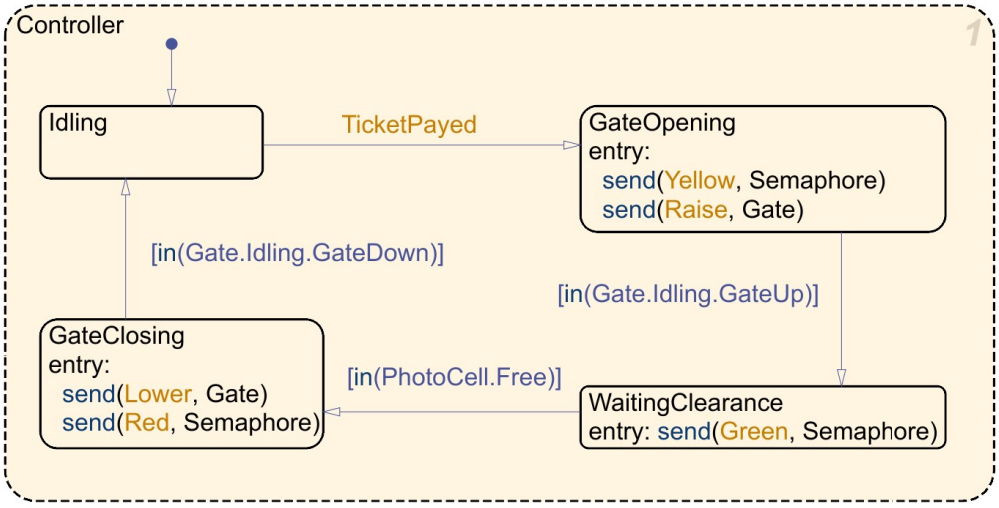

Figure 5: Control Block

Gate As already mentioned, the *Gate* block is responsible for controlling the gate's position and ensuring that it opens and closes correctly. It's designed based on two hierarchical states: *Idling* and *Moving*, which then are further divided into *GateUp* and *GateDown* states, and *Raising* and *Lowering* states, respectively. At first, the *Gate* block is in the *GateDown* state, where the gate is closed and the vehicle is not allowed to pass through. Once the *Raise* event is received from the *Controller* block, the *Gate* block moves to the *Raising* state, where it starts to open the gate. Once the gate is fully opened, the *Gate* block moves to the *GateUp* state waiting for

the *Lower* event from the *Controller* block that will then perform the same steps in reverse order, moving to the *Lowering* state and then to the *GateDown* state once the gate is fully closed.
Figure 6 shows the *Gate* block of the parking gate FSM.

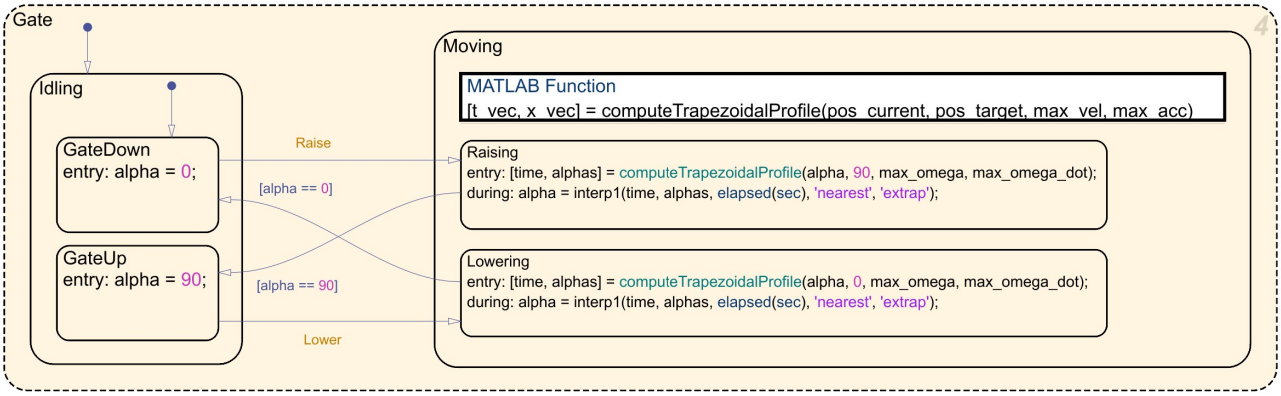

Figure 6: Gate Block

Notice that, in order to respect the constraints $w_{max} = |4|[^\circ/s]$ and $\dot{w}_{max} = |1|[^\circ/s^2]$, the position of the gate is updated based on the trapezoidal motion profile, generated when entering the *Raising* or *Lowering* states. The trapezoidal motion profile is defined by the following equations:

$$w(t) = \begin{cases} \frac{w_{max}}{t_1} t & 0 \leq t < t_1 \\ w_{max} & t_1 \leq t < t_2 \\ w_{max} - \frac{w_{max}}{t_3}(t - T) & t_2 \leq t < T \end{cases} \tag{1}$$

Where t_1 is the time when the gate reaches the maximum speed and t_2 is the time when the gate starts to decelerate. The trapezoidal motion profile is designed to ensure that the gate opens and closes smoothly, without any sudden jerks or stops. Listing 1 shows the implementation of the trapezoidal motion profile as MATLAB function.

```
function [t_vec, x_vec] = computeTrapezoidalProfile(pos_current, pos_target, max_vel,
    max_acc)
% COMPUTE_TRAPEZOIDAL_PROFILE Generates time and angle points for a trapezoidal profile
% At first checks if a trapezoidal profile is feasible or is necessary to
% fallback to a triangular one.

delta_pos = pos_target - pos_current;
time_acc_phase = max_vel / max_acc;
delta_pos_acc_phase = 0.5 * max_acc * time_acc_phase^2;

if 2 * delta_pos_acc_phase < abs(delta_pos)
    % Trapezoidal profile
    t1 = time_acc_phase;
    t2 = (abs(delta_pos) - 2 * delta_pos_acc_phase) / max_vel;
    t3 = time_acc_phase;
else
    % Triangular profile
    max_vel = sqrt(abs(delta_pos) * max_acc);
    t1 = max_vel / max_acc;
    t2 = 0;
    t3 = max_vel / max_acc;
end


% Trapezoidal profile template (HP: s0 = 0, v0 = 0)
acc = max_acc * sign(delta_pos);
vel = max_vel * sign(delta_pos);
profile_template = @(t) ...
    (t <= t1) .*                              (1/2 * acc * t.^2) + ...
```

```
    (t > t1 & t <= (t1 + t2)) .*              (1/2 * acc * t1^2 + vel * (t - t1)) + ...
    (t > (t1 + t2) & t <= (t1 + t2 + t3)) .* (1/2 * acc * t1^2 + vel * t2 + vel * (t - (
    t2 + t1)) - 1/2 * acc * (t - (t1 + t2)).^2) + ...
    (t > (t1 + t2 + t3)) .*                   (1/2 * acc * t1^2 + vel * t2 + vel * t3 -
    1/2 * acc * t3.^2);

% Profile generation
t_vec = linspace(0, t1 + t2 + t3, 100);
x_vec = profile_template(t_vec) + pos_current;

end
```

Listing 1: Trapezoidal Motion Profile

PhotoCell Despite being used only during the *WaitingClearance* state of the *Controller* block, the *PhotoCell* block is responsible for detecting the vehicle's presence in front of the gate. This, in fact, is the only way to understand if the vehicle has passed through the gate and the gate can be closed again safely. In order to continuously sample the vehicle's position, the *PhotoCell* block is designed to not be event-driven, but to move continuously between the two states *Free* and *Occupied* based on the vehicle's position. By doing so, an external agent can continuously monitor the vehicle's position by checking the state of the *PhotoCell* block. In order to determines the presence or not of the vehicle, the *PhotoCell* block uses the knowledge of the vehicle's length and position, the latter of which is shared by the *Vehicle FSM*.
Figure 7 shows the *PhotoCell* block of the parking gate FSM.

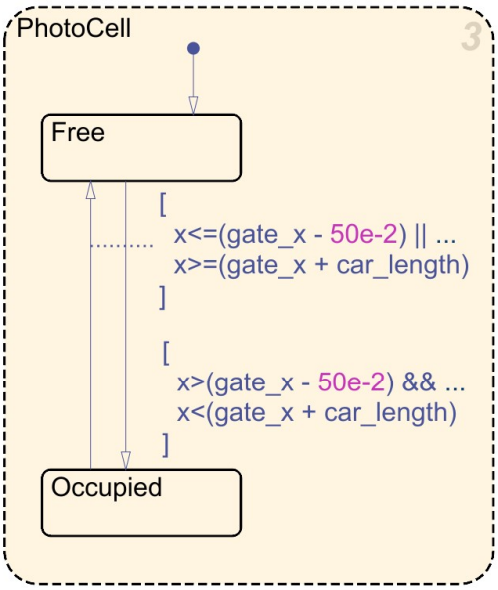

Figure 7: PhotoCell Block

Semaphore Last but not least, the *Semaphore* block is responsible for switching the semaphore state between red, yellow and green based on the events received from the *Controller* block. It has one single state *On* composed of three sub-states representative of the color of the semaphore. Based on the event received, the correct sub-state is chosen and the semaphore state is then shared with the *Vehicle FSM* to inform it about the state of the gate (open, closed, etc.). This is done in order to inform the vehicle about the state of the gate and allow it to proceed or stop based on the semaphore state, which is the only information that the vehicle has about the parking gate system (i.e. the vehicle doesn't know if the gate is open or closed, but only if the semaphore is red or green).
Figure 8 shows the *Semaphore* block of the parking gate FSM.

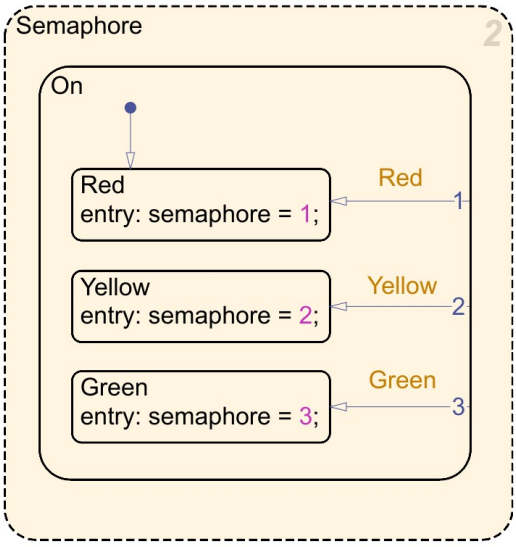

Figure 8: Semaphore Block

3.3 Simulation Results

The simulation has been run for 100s and the vehicle has been reset to a new random position right after the closure of the gate. The following figures show the results of the simulation.

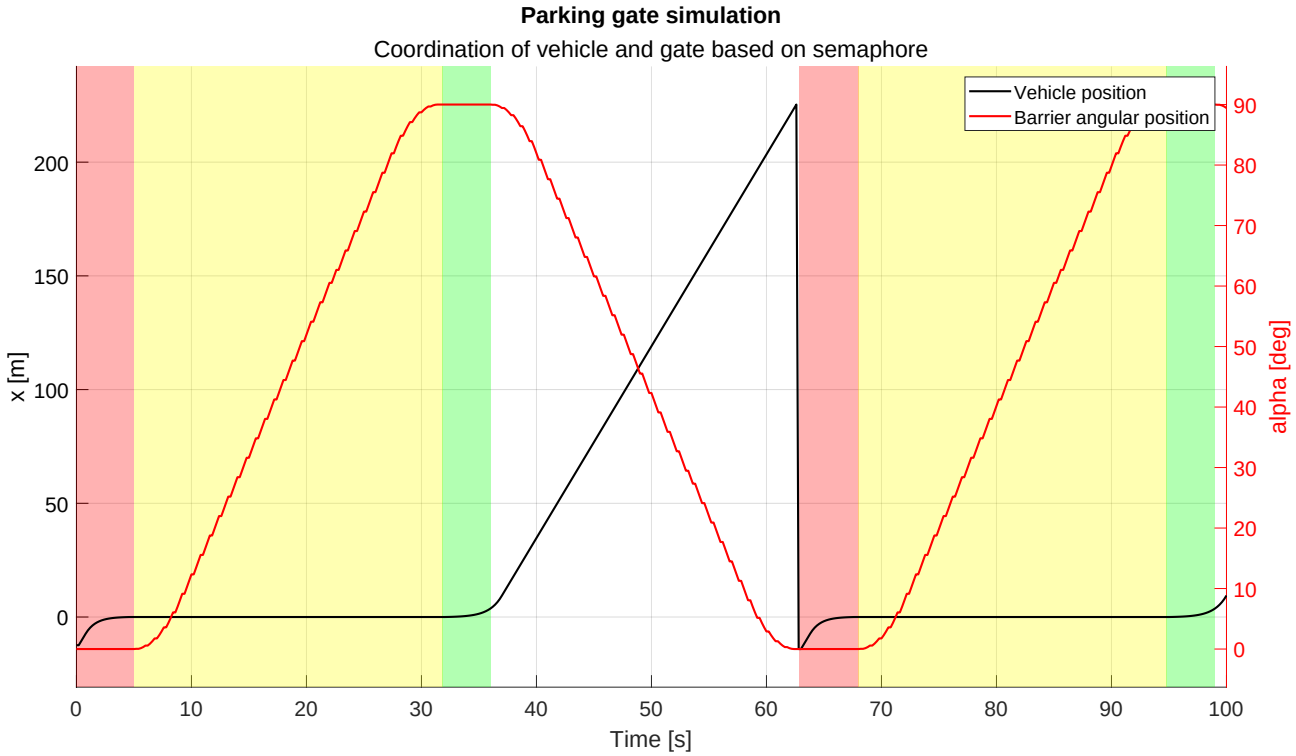

Figure 9: Simulation Results

Figure 9 shows the vehicle position (in black) and the gate angular position (in red) over time. The plot has also been divided into different regions based on the state of the *Semaphore* block (red, yellow and green). Notice that for clarity, the semaphore state has been added to the plot only when actually visible by the vehicle (i.e. when the vehicle is in front of the gate and not yet passed through it).
Instead, a more detailed view of the underlying coordination between the two FSMs is shown in Figure 10, where a visualization of the state transitions and events is shown. The plot report on the vertical axis the time of the simulation and on each vertical dashed line the state of each superstate of the FSMs is shown. One can appreciate how the two FSMs are coordinated and how the events are sent and received between them, along with the transitions between the states.

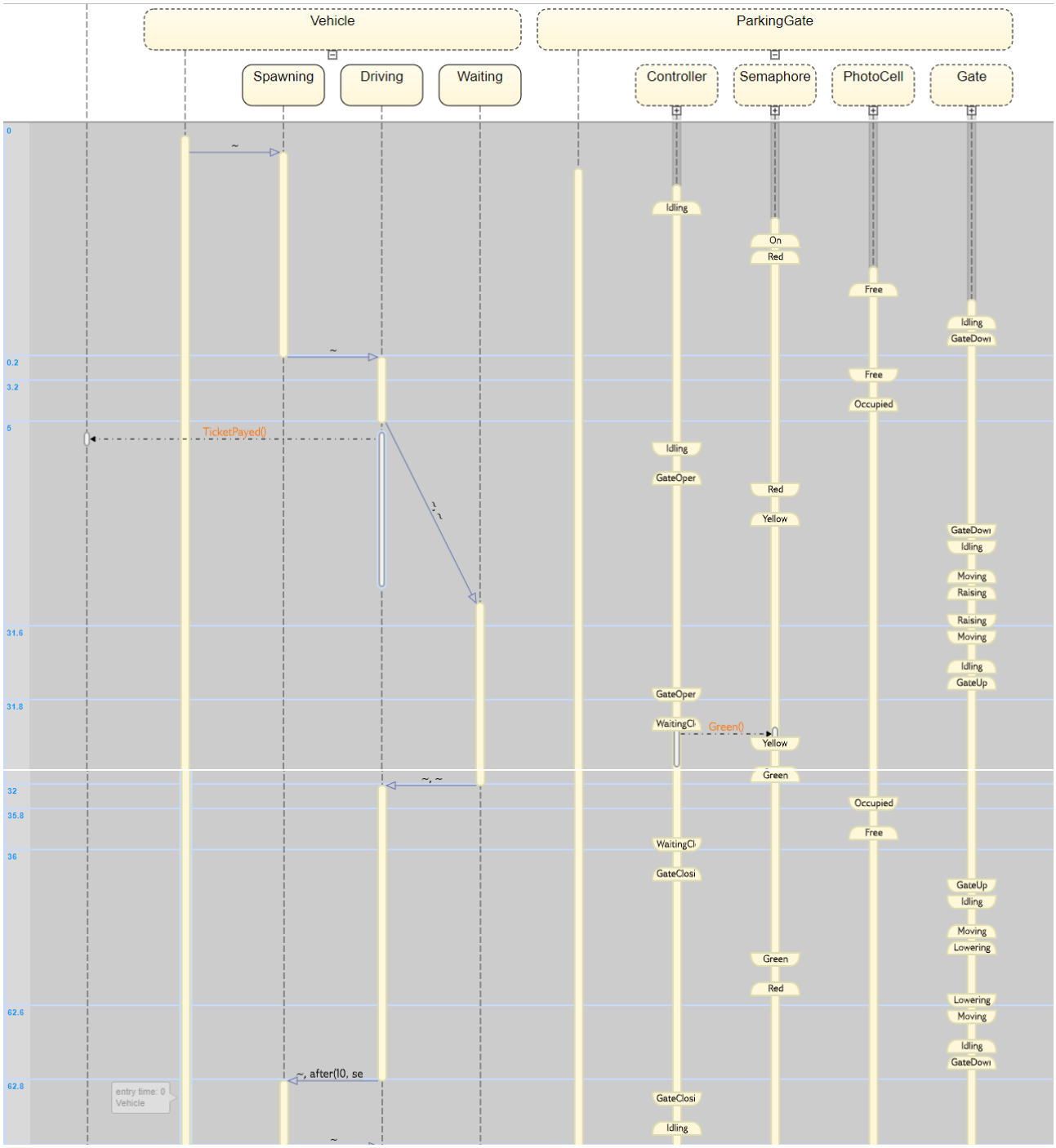

Figure 10: Simulation Results - Detailed View

4 Conclusions

In this short report, we have presented the results of the fifth assignment of the course on Autonomous Vehicles. We have briefly explained what FSMs are and how they can be used to model the behavior of generic systems. We have also presented a possible FMSs implementation about a car parking system, which is able to coordinate the behavior of the car and the parking gate. The results presented have shown the effectiveness of the proposed solution.

The current solution could be improved under many aspects. For example, one could think of implementing features like error handling, or ticket authentication and validation, or even a more complex parking system composed of multiple parking gates and cars that must coordinated over the network.

We believe that FMSs are a power tool that can often replace traditional programming paradigms, allowing to model complex systems in a more intuitive way. Further studies on this topic could be useful to understand the full potential of FMSs and their applications in the field of autonomous vehicles and robotics in general.

References

[1] Wikipedia contributors. Finite-state machine — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Finite-state_machine&oldid=1285525354`, 2025. [Online; accessed 26-April-2025].

A Appendix

The system presented in Section 3.2, leverages one single FSM including under the hood both the vehicle and the parking gate FSMs. However, this solution is not the most elegant one given that the two FSMs are actually two separate entities that communicate with each other via external signals. For this reason, the first solution and implementation of the parking gate system was the one shown in Figure 11.

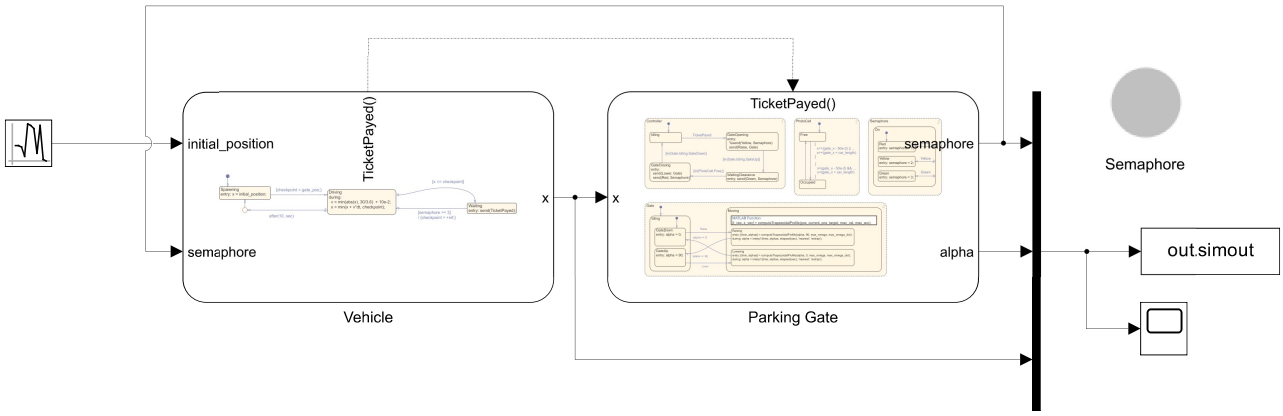

Figure 11: Original Parking Gate System Overview

Notice that the underling components and states are exactly the same as the ones presented in the previous section. Nonetheless, the structure of Figure 11 resolved in unexpected behavior of the system and never generated the expected results.
Even with the help of debugging tools, it was impossible to understand the reason behind the unexpected behavior of the system and the author had to simplify the structure and fallback to the single FSM solution discussed during the document (compromise between elegance and functionality).