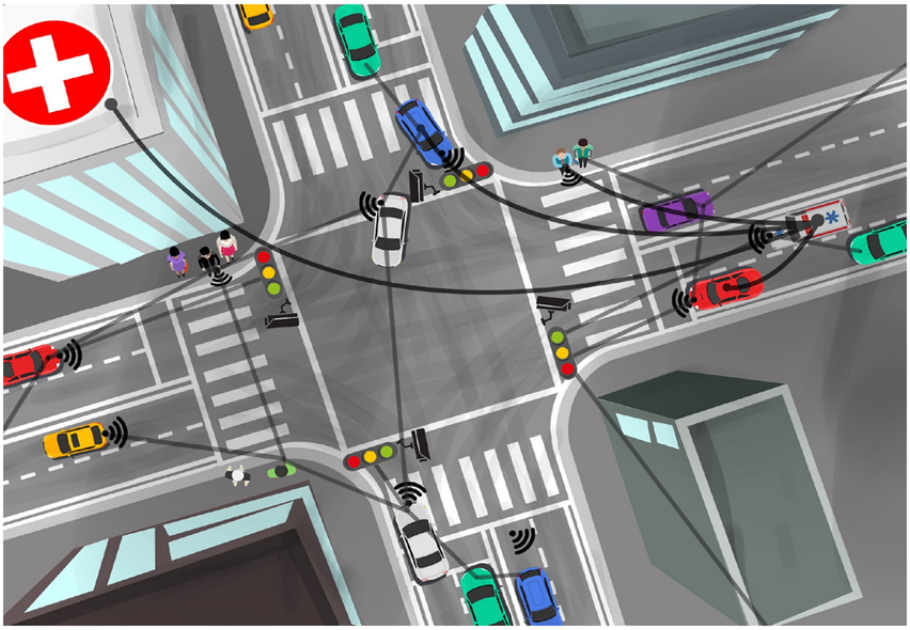
Smart Crossroads

Intelligent Systems Engineering



Emanuele Lamagna

March 2025

Contents

Chapter 1

Introduction

Smart Crossroads is an application that aims to simulate traffic in a certain area of a city, in which there are 4 intersections regulated by traffic lights. The goal of the project is to exploit agent programming, provided by the combination of **Java** and **Jason**, to implement efficient traffic management through communication between intelligent agents, i.e. cars and traffic lights.

The vision of the project is that of an automatic management of cars within an area: a self-driving car, for example, may want to communicate with traffic lights and other cars to create intelligent traffic management, calculate better routes, save fuel/energy, etc. This is still a fairly unrealistic scenario today, but it certainly represents an interesting idea regarding what an automatic traffic management could be like through various agents of various types interconnected with each other.

Chapter 2

Requirements

The functional and non-functional requirements of the project included:

2.1 Functional requirements

- the cars should spawn and follow a path
- the traffic lights should follow a cycle between them (divided by cross-roads). The duration of the traffic light green color should be variable, according to the presence or not of cars in the road of that traffic light
- the cars should stop if the corresponding traffic light is not green
- the cars should stop if there is another car in the the target position (so, it should stop and wait until that position is free)
- the cars should disappear once the path is closed
- the cars shouldn't randomly stop while traveling on the roads

2.2 Non-functional requirements

- the system has to be fluid: the cars shouldn't stop in every tile too much to think if it can proceed
- the simulation should avoid long car queues
- the simulation should have a monitoring panel, with clear logs

2.3 Technologies

This project includes the following techologies:

- **Jason**, an interpreter for an extended version of AgentSpeak, a BDI agent-oriented logic programming language, used to model and manage intelligent agents.
- **Java**, used for the application logic and system integration. The main choice was between Java and Kotlin, but the first was chosen for ease of use and habit, also because it is not a project that requires the advanced features that Kotlin can provide better.
- **JavaFX**, chosen for the development of the graphical interface, ensuring an intuitive user experience. Compared to Swing, it is certainly a more complex solution in terms of programming and integration, but also much more modern and captivating.
- **JUnit5**, to test the model of the agents

The combined use of these technologies allowed to create a robust and modular system, capable of managing the complexity of the agents' decisions and of clearly presenting the information to the user.

2.4 Necessary changes

The initial proposal for the project also included a further aspect, that of fines: a car should have a small probability of passing through a red light, and in that case receive a fine. All this would have resulted in a message to a control agent (central station agent) who would have done nothing but send a log of the fine to the monitoring panel. Although it was a rather simple task (it was a message between agents and a printout on the panel) it would have led to a big problem related to the tiles. The graphics, in fact, were designed in a simple way, to focus on the communication between agents, and in fact the center of a crossroads is made up of 4 tiles. In each crossroads only one traffic light turns green at a time, so we do not have problems of concurrency or various deadlocks. However, if a car passed through a red light, an error situation could occur, simply due to the number of tiles set. In fact, in a situation where a car passes (legitimately) with the green light to go left and another car from the opposite direction passes with the red light (not legitimately) wants to do the same, a deadlock occurs: the cars want to occupy the tile of the other, which however will not free itself. The solution would have involved:

- increasing the number of tiles
- doubling the lanes for each direction of travel

But by doing so the project hours dedicated to the graphics part would have become many (it should be specified that the hours planned for the project were reached anyway). It was therefore decided not to implement this feature, simply hypothesizing how it could be hypothetically solved.

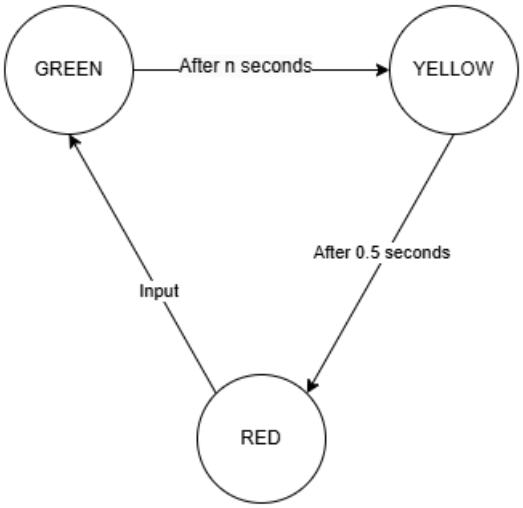
Chapter 3

Design

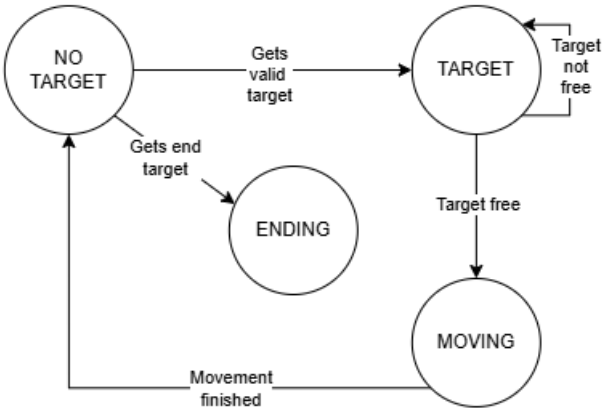
3.1 Agents

The system requires that every n seconds a car appears and follows a path through the streets. The cars, once they reach the intersections, will have to check the traffic light to understand whether they can pass or stop. Once the path is finished, the car leaves the area. Trying to abstract the problem, it was decided to create three types of agents:

- **car agent**: it is the agent that manages the car, its route and the communication with the traffic lights and with the other cars (in fact, to avoid discounts or overlaps, the cars must also understand if there is already someone else in the "target" position)
- **traffic light agent**: it is the agent who manages the traffic light. The traffic lights are divided by intersections: each intersection has 4 traffic lights, one of which starts green and the others start red. At each intersection only one traffic light is green at a time, for a variable duration depending on the number of cars waiting. Then it turns yellow, then red and finally it tells the next traffic light to turn green.
- **creator agent**: it is the agent that creates the other agents. It initially creates the 16 traffic lights (4 traffic lights at each of the 4 intersections), and then starts creating the cars every n seconds.



Traffic light agent

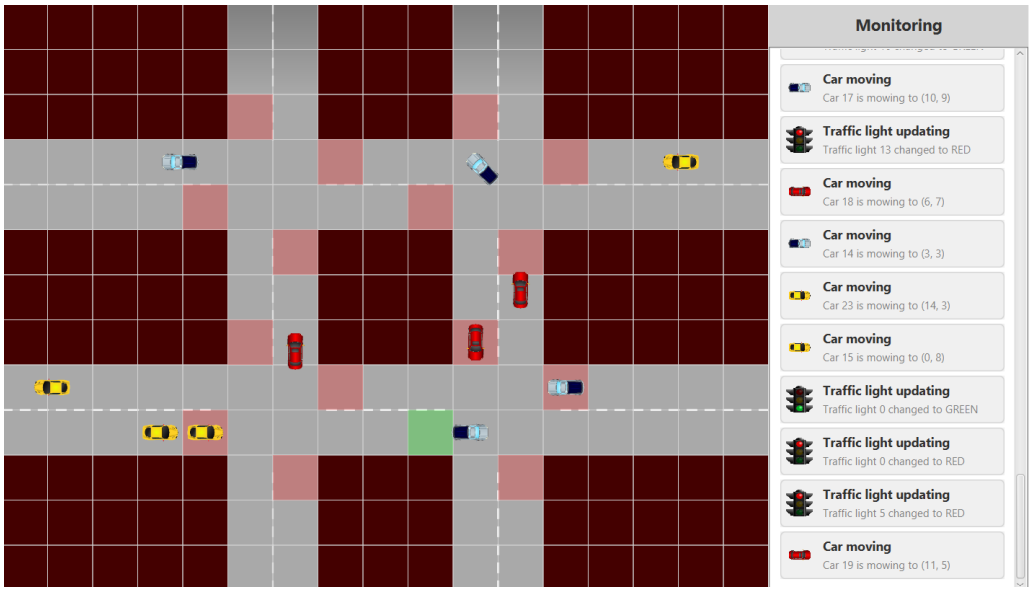


Car agent

3.2 Environment

It was necessary to use an environment to manage the communications from the agents to the logic and graphics. an agent model (in Java) was created to abstract them and better manage the agents' perceptions (for example the target calculation). This model then contains all the agents' data, exploiting the mechanisms of OOP, which are then translated into perceptions when requested by the various agents.

3.3 UI



As mentioned, the graphics were created with JavaFX. There are four roads, which intersect and form the 4 intersections. Each road has two lanes, which have been logically divided into tiles. Each car can occupy one tile at a time, and the movement takes place from tile to tile. The graphics are rather minimal, with just the necessary indications such as the colors of the road, traffic lights and icons of the cars that move. In addition to the main screen we also have a panel on the right, which reports all the events of the simulation (creation and movement of cars, exit, change of color of traffic lights).

Chapter 4

Implementation details

4.1 Agents

We have talked about the three types of agents that are part of the system, and what they do in broad terms. Let's now analyze their behaviors in more detail.

4.1.1 Creator agent

The creator agent is responsible for creating traffic lights and cars. In order:

- Creates the 16 traffic lights with a loop. The traffic light is created using the action .create_agent(name, agent_file). Then the values of number (the identification number from 0 to 15) and type (the type of traffic light, i.e. a number from 0 to 3 that identifies the position of the traffic light with respect to the intersection) are passed using *.send(name, action, belief)*, with the *tell* parameter as the second parameter. Finally, it takes care of having the traffic light agent execute the startup plan with another *.send*, this time specifying *achieve* as the action and passing its position.

```
+!start_creation <-
    List1 = [5, 6, 4, 7, 10, 11, 9, 12, 10,
        11, 9, 12, 5, 6, 4, 7];
    List2 = [2, 5, 4, 3, 2, 5, 4, 3, 7, 10,
        9, 8, 7, 10, 9, 8];
    while(lights(L) & L <= 15) {
        .nth(L, List1, PosX);
        .nth(L, List2, PosY);
```

```
            .concat("traffic_light_", L, N);
            .create_agent(N, "traffic_light_agent.asl");
            TYPE = L mod 4;
            .send(N, tell, number(L));
            .send(N, tell, type(TYPE));
            .send(N, achieve, start(PosX, PosY));
            -+lights(L + 1);
        };
        !create_next_car.
```

- After that creates, every n, seconds, a car. It calculates a random starting position (one of 8 possible) and then creates the car agent via the usual *.create_agent()*. It then sends the *achieve* with the car position, which can start its execution flow.

```
+!create_next_car : cars(C) <-
    .random(R);
    RInt = (R * 8) - ((R * 8) mod 1);
    !get_spawn_position(RInt, PosX, PosY, D);
    .concat("car_", C, N);
    .create_agent(N, "car_agent.asl");
    .send(N, achieve, start(PosX, PosY, D));
    -+cars(C + 1);
    .wait(2000);
    !create_next_car.
```

4.1.2 Traffic light agent

There are 16 traffic lights in total, 4 for each intersection. Each traffic light starts when it receives an *achieve* from the creator agent. Once started, it checks if it is green or not (initially traffic lights with type = 0, i.e. those to the north, start green), then communicates its spawn to the environment, which will take care of updating the graphics. If it starts green, it waits two seconds and then turns yellow, otherwise it does nothing.

```
+!start(PosX, PosY) : type(T) <-
    .my_name(Me);
    spawn_traffic_light(T = 0, PosX, PosY, Me);
    if (T = 0) {
```

```
        .wait(2000);
        !cycle(yellow);
    }.
```

Then follows a cycle:

- when it turns green, it communicates this to the environment and then checks how many cars it has in the queue: if it has none, it goes directly to red, otherwise it remains green for a number of seconds equal to the number of cars in the queue.

```
+!cycle(green) <-
    .my_name(Me);
    update_traffic_light(green, Me);
    .count(position(_, _), N);
    if (N = 0) {
        !cycle(red);
    } else {
        TIME = N * 1000;
        .wait(TIME);
        !cycle(yellow);
    }.
```

- when it turns yellow, it communicates it to the environment and then shortly after it turns red

```
+!cycle(yellow) <-
    .my_name(Me);
    update_traffic_light(yellow, Me);
    .wait(500);
    !cycle(red).
```

- when it turns red, it tells the environment and then checks which traffic light is next. Once it calculates it, it sends an *achieve* to that traffic light to tell it it can go back to green.

```
+!cycle(red) : number(N) <-
    .my_name(Me);
```

```
        update_traffic_light(red, Me);
        .wait(1000);
        if (not(N = 0) & N mod 4 = 3) {
            M = N - 3;
        } else {
            M = N + 1;
        };
        .concat("traffic_light_", M, Res);
        .send(Res, achieve, cycle(green)).
```

How does a traffic light get the positions of the cars it is interested in? As we will see in the next subsection, cars broadcast their positions every time they move. Traffic lights receive these messages and, if the position matches the road they cover, they save it.

```
+!share(PosX, PosY)[source(Other)] : (Other \== self) &
  type(T) & tl_position(X, Y) & number(N) <-
    -position(_, _)[source(Other)];
    if (T = 0 & PosX = X &
        (PosY = Y | (PosY + 1) = Y | (PosY + 2) = Y)) {
            +position(PosX, PosY)[source(Other)];
    };
    if (T = 1 & PosX = X &
        (PosY = Y | (PosY - 1) = Y | (PosY - 2) = Y)) {
            +position(PosX, PosY)[source(Other)];
    };
    if ((N = 2 | N = 14) & PosY = Y
        & (PosX = X | (PosX + 1) = X | (PosX + 2) = X |
        (PosX + 3) = X | (PosX + 4) = X)) {
            +position(PosX, PosY)[source(Other)];
    };
    if ((N = 7 | N = 11) & PosY = Y &
        (PosX = X | (PosX - 1) = X | (PosX - 2) = X |
        (PosX - 3) = X | (PosX - 4) = X)) {
            +position(PosX, PosY)[source(Other)];
    };
    if ((N = 3 | N = 15) & PosY = Y & (PosX = X |
        (PosX - 1) = X | (PosX - 2) = X)) {
            +position(PosX, PosY)[source(Other)];
    };
```

```
if ((N = 6 | N = 10) & PosY = Y & (PosX = X |
    (PosX + 1) = X | (PosX + 2) = X)) {
        +position(PosX, PosY)[source(Other)];
}.
```

4.1.3 Car agent

A new car is created every n seconds by the creator agent. When a car agent starts its execution flow, it receives its initial position and direction from the creator, then sends a *broadcast* message to all cars (specifying *askOne* as the action) to ask for the position of each one and save it. It then tells the environment about the spawn (the environment will tell the graphics).

```
+!start(PosX, PosY, D) <-
    .my_name(Me);
    .broadcast(achieve, ask_position);
    spawn_car(PosX, PosY, Me, D).
```

When a car then receives a message inviting it to share its location, it communicates this to the sender:

```
+!ask_position[source(Other)] : (Other \== percept)
  & position(X, Y)[source(percept)] <-
    .send(Other, tell, position(X, Y)).
```

Every time the environment perceives that a car's action is finished (the spawn or the move) it tells the model to calculate the new target position (we'll see this in the next section). The car agent, therefore, finds itself with a new target belief, which it must manage based on traffic lights and other cars. Going in order:

- when the target arrives, the specific plan is executed:

  ```
  +target(PosX, PosY) <-
      !check_target(PosX, PosY).
  ```

- if the position is already occupied by a machine, I try again until the position becomes free:

```
+!check_target(PosX, PosY) : position(PosX,
  PosY)[source(Other)] & (Other \== percept) <-
    !check_target(PosX, PosY).
```

- if the position is free, it checks if there is a traffic light. If there is not, then it moves, otherwise it checks the traffic light by sending a *send* message with *askOne* action to the traffic light in question. If it responds that it is green, it proceeds, otherwise it stops (the list of traffic lights comes from perceptions):

```
+!check_target(PosX, PosY) : not(position(PosX,
  PosY)[source(Other)]) & (Other \== percept) &
  position(X, Y)[source(percept)] <-
    if (tl(S, X, Y)) {
        .concat("traffic_light_", S, TL);
        .send(TL, askOne, is_green(GREEN),
            is_green(GREEN));
        if (not(GREEN)) {
            !check_target(PosX, PosY);
        } else {
            !go(PosX, PosY);
        }
    } else {
        !go(PosX, PosY);
    }.
```

- if the car receives a target but is in position (-1, -1), therefore outside the map, then it broadcasts its termination with the untell action (to remove the belief of its position from other cars) and terminates, communicating first to the environment and then with .kill_agent(name):

```
+!check_target(PosX, PosY) : PosX = -1 & PosY = -1
  & position(X, Y)[source(percept)] <-
    .broadcast(untell, position(X, Y));
    !terminate.

+!terminate : name(Me) <-
    remove_car(Me);
    .kill_agent(Me).
```

- if a car moves, it simply updates its position, broadcasts its new position to everyone, who then proceeds to save it in their beliefs:

```
+!go(PosX, PosY) : direction(D) & name(Me) <-
    .broadcast(achieve, share(PosX, PosY));
    move_car(PosX, PosY, Me, D).

+!share(PosX, PosY)[source(Other)] :
  (Other \== percept) <-
    -+position(PosX, PosY)[source(Other)].
```

4.2 Environment and model

Traffic light agents and car agents are abstracted via the TrafficAgent abstract class, which is then extended by TrafficLightAgent and CarAgent respectively. The TrafficAgent class is defined as follows:

```
public abstract class TrafficAgent {
    abstract Set<Literal> getPercepts();
}
```

CarAgent and TrafficLight will have their own fields, with their own getters and setters. The most important thing is the extension of the getPercepts method, which agents use all the time to get perceptions:

- TrafficLightAgent

```
@Override
Set<Literal> getPercepts() {
    final Set<Literal> set = new HashSet<>();
    final int x = position.getFirst();
    final int y = position.getSecond();
    set.add(Literal.parseLiteral(
        String.format("tl_position(%d, %d)", x, y)));
    set.add(Literal.parseLiteral(
        String.format("name(%s)", name)));
    set.add(Literal.parseLiteral(
        String.format("is_green(%s)", isGreen)));
```

```
        return set;
    }
```

- CarAgent

```
    @Override
    public Set<Literal> getPercepts() {
        final Set<Literal> set = new HashSet<>();
        final int x = position.getFirst();
        final int y = position.getSecond();
        set.add(Literal.parseLiteral(
            String.format("position(%d, %d)", x, y)));
        set.add(Literal.parseLiteral(
            String.format("name(%s)", name)));
        if (direction != null) {
            set.add(Literal.parseLiteral(
                String.format("direction(%d)",
                direction.ordinal()
            )));
        }
        if (target != null) {
            final int targetX = target.getFirst();
            final int targetY = target.getSecond();
            set.add(Literal.parseLiteral(
                String.format("target(%d, %d)",
                targetX,
                targetY
            )));
        }

        for (int i = 0; i < trafficLights.size(); i++) {
            var tl = trafficLights.get(i);
            set.add(Literal.parseLiteral(
                String.format("tl(%d, %d, %d)",
                i,
                tl.getFirst(),
                tl.getSecond())));
        }

        return set;
```

```
        }
```

The model then has the following methods:

```
public interface TrafficModel {
    Map<String, TrafficAgent> getAllAgents();

    void insertAgent(
        String name,
        TrafficAgent agent
    );

    TrafficAgent getAgent(String name);

    void removeAgent(String name);

    Set<Literal> getPercepts(String agent);

    void moveCar
        (String name,
        Pair<Integer, Integer> position,
        int dire
    );

    void updateTrafficLight(
        String name,
        LightColor color
    );

    void calculateTarget(String name);
}
```

As for the TrafficEnvironment, it contains the reference to the model and the view. It extends, like any Jason environment, the methods *init* (where the model is initialized), *getPercepts* (which returns the perceptions of a certain agent), *executeAction* (which is triggered when an agent performs an operation on the environment, for example *spawn_car*):

```
@Override
public void init(final String[] args) {
```

```
    model = new TrafficModelImpl();
}

@Override
public Set<Literal> getPercepts(String agName) {
    return this.model.getPercepts(agName);
}

    @Override
public boolean executeAction(
    final String ag,
    final Structure action
) {
    ...
    switch (actionName) {
        case Actions.SPAWN_CAR:
            ...
            this.model.insertAgent(...);
            notifyCarSpawned(...);
            ...
        case Actions.SPAWN_TRAFFIC_LIGHT:
            ...
            this.model.insertAgent(...);
            notifyTrafficLightSpawned(...);
            ...
        case Actions.UPDATE_TRAFFIC_LIGHT:
            ...
            this.model.updateTrafficLight(...);
            notifyTrafficLightUpdate(...);
            ...
        case Actions.MOVE_CAR:
            ...
            this.model.moveCar(...);
            notifyCarMoved(...);
            ...
        case Actions.REMOVE_CAR:
            ...
            this.model.removeAgent(...);
            notifyCarRemoved(...);
            ...
        default:
```

```
            ...
        }
    }
```

The various notify methods are used to communicate certain actions (spawn, movements, etc.) to the UI.

4.3 UI

The Launcher is the main class of the application, responsible for initializing and managing the traffic simulation. When started, it loads the necessary images for cars and traffic lights, sets the window dimensions based on the screen resolution, and initializes the road grid with the corresponding Tiles. Then, it launches a Jason multi-agent environment for traffic management, executing the MAS configuration file (crossroads.mas2j) in a separate thread. After initialization, the graphical interface is built using JavaFX, including a canvas for rendering cars and roads, as well as a sidebar for monitoring traffic events. The system updates the graphics in real-time using a Timeline, redrawing the scene at each frame to display car movements and traffic light states. The Launcher implements the TrafficListener interface, allowing it to receive updates from the system about:

- car spawn:

```
synchronized (cars) {
    int randomType = new Random().nextInt(3) + 1;
    cars.add(new Car(carId, randomType, posX, posY));
}
```

- car movement:

```
synchronized (cars) {
  for (var car : cars) {
    if (car.getId() == carId) {
      ScheduledExecutorService scheduler =
        Executors.newSingleThreadScheduledExecutor();
        scheduler.scheduleAtFixedRate(() -> {
          boolean finished = car.move(posX, posY);
          if (finished) {
```

```
                environment.notifyAnimationFinished(
                  carId,
                  posX,
                  posY,
                  Direction.values()[dir]);
                  car.setPosition(posX, posY);
                  scheduler.shutdown();
              }
            }, 0, 30, TimeUnit.MILLISECONDS);
          break;
        }
      }
    }
```

This is the most complicated case. It was necessary to make sure that, once the movement was finished, the environment was notified, so that a new target could be assigned to the agent and it could be restarted.

- traffic light spawn:

```
synchronized (tiles) {
    for (var tile : tiles) {
        if (tile.getPosX() == posX &&
          tile.getPosY() == posY) {
            tile.setTrafficLightId(trafficLightId);
            tile.setColor(isGreen ?
              LightColor.GREEN :
              LightColor.RED);
        }
    }
}
```

- traffic light update:

```
synchronized (tiles) {
    for (var tile : tiles) {
        if (tile.getTrafficLightId() != -1 &&
          tile.getTrafficLightId() == trafficLightId) {
```

```
                tile.setColor(color);
            }
        }
    }
```

- car removal

```
synchronized (cars) {
    cars.removeIf(car -> car.getId() == carId);
}
```

Chapter 5

Conclusion

This project has been an incredibly valuable learning experience, as it allowed me to work with a type of programming that I had only briefly encountered in other courses. Throughout the development process, I had to face a variety of complex and non-trivial challenges, which pushed me to think critically and find creative solutions. These difficulties not only strengthened my problem-solving skills but also deepened my understanding of real-time simulations and multi-agent systems.

Another key aspect of this project is its high expandability. There are countless features that could be added to make the simulation even more detailed and realistic. For example, implementing pedestrian crossings would introduce new interactions between vehicles and pedestrians, while adding overtaking mechanics would make traffic flow more dynamic and unpredictable. Speed limits, traffic signs, and other regulatory elements could further enhance the realism of the system. Naturally, such improvements would require a larger and more complex map, but this would also open the door to even more sophisticated behaviors and interactions, making the simulation an even more powerful tool for studying and analyzing traffic dynamics.