# Smart Assistant Drive

**Intelligent Systems Engineering**

Andrea Brighi
andrea.brighi8@studio.unibo.it

Daniele Di Lillo
daniele.dilillo@studio.unibo.it

January 14, 2026

From July 2024, all newly produced cars must have a camera to read speed signs (Intelligent Speed Assist), but this technology has yet to be ready as we expect it to be. As shown in this Quattroruote test , only some signs can be read correctly. This is because street signs are a human concept for human comprehension; in our opinion, cars should use a digitalized version of them. This also expands to other road-related concepts, such as traffic lights. The idea is to simulate a road and apply MAS to cars and traffic lights, representing them as artifacts controlled by agents that interact with them.

## 1 Goal/s of the project

The project's main goal is to explore new technologies and apply them for a new smart drive assistant paradigm, to evaluate benefits and use cases for Multi-Agent Systems and Digital Twins.

- Create a digital version of a map composed of streets, intersections, and traffic lights
- Use an Agent to control each junction with traffic lights
- Use an agent to simulate the car that helps the human driver (or replaces him), understands the signs on the road, and regulates the speed to reduce fuel consumption (Moderate speed or slowing, considering the color of the traffic lights).
- Improve car agents to autonomously drive from point A to point B, by understanding signs, traffic lights, and traffic flow.

### 1.1 Usage scenarios

The system will be a global traffic simulation, composed of roads, cars, signals, and traffic lights for junctions. By this way, we expect to start the simulation using a specific scenario describing the number of elements to spawn and preparing them to execute independently.

As the simulation runs, we expect to see cars moving on their paths between the established start and end points, and we'll be able to see how they will behave in the traffic jam, with road signals (that might change the driving rules), junctions, and traffic lights in a completely autonomous way.

## 2 Background and link to the theory

- Relevant architectural styles
  - Inspired to Agents & Artifacts (A&A) conceptual model
  - Layered Architecture (Distributed)
- Relevant software frameworks
  - White Label Digital Twins paper - WLDT framework
- Relevant Math concept
  - polyline

## 3 Requirements Analysis

Let us dig into all the requirements in order to accomplish all the project goals.

### 3.1 Functional requirements list

- Roads, junctions, and road signs models should be managed in repositories
- The simulation should start from a predefined scenario, starting all the necessary services and populating the databases with initial information
- The cars should start from a point and follow a path for their trip to go from the starting point A to final point B, in a shared environment with other cars and junctions, and not create crashes or violate the road laws (eg, speed limit excess, red light run)
- The simulation should show all the elements, such as roads, cars, traffic lights, and signs
- Every single car must keep a security distance from the next one
- Every junction is regulated by a semaphore, including 4 lights (one for each direction), and must be temporized

### 3.2 Non-functional requirements list

- The simulation should be human comprehensible
- The update message from the traffic light to the car should have an acceptable delay (no more than 0.5 seconds)

### 3.3 Implementation requirements list

- Track the cars' information into a specific traffic Digital Twin, to calculate the distances and manage the traffic queues
- Track the semaphore states and events into a Digital Twin
- Use agents to model the behavior of cars and coordinate the junction's traffic lights

### 3.4 Assumptions

- GPS for cars and Internet connection for all elements
- The cars have radar for near collision (simulated in the simulation with the traffic Digital Twin)
- Lane assist and a camera to recognize road lanes (abstract lane as a discrete space and not a continuous space)
- There are only cars moving on the street (no other vehicle or person)

### 3.5 Technologies

- **Java**: it's the main language for the agent's environment definition, logics, and behaviors
- **Jason**: is an interpreter for an extended version of AgentSpeak. It implements the operational semantics of that language and provides a platform for the development of multi-agent systems, with many user-customisable features. It's the building tool for the multi-agent system.
- **WLDT**: White Label Digital Twin is the framework used to develop the Digital Twins that feed the agents with the sensing information and provide the middleware between the physical entities and the intelligent agents
- **Kotlin**: it's the second programming language of the project, used for the microservices and Digital Twin development
- **Kotlin Compose**: this framework is used to build beautiful shared UIs for Android, iOS, desktop, and web that feel natural on every platform

# 4 Design

The design of the Smart Road infrastructure is centered around a Multi-Agent System (MAS), where autonomous entities interact to optimize traffic safety, flow, and efficiency. In this model, agents are not merely passive components but proactive decision-makers capable of perceiving the environment and acting upon it to achieve specific goals. To accomplish the project goals, we have defined the following Multi-Agent Systems:

- Car MAS: it manages a car agent in the simulation, has limited vision, and perceives the environment using the microservice (road structure and sign) and MQTT broker (connected to the Digital Twin) to get updates about traffic lights and the car ahead (not known by the agent)
- Junction MAS: it manages all the semaphore agents in the simulation

By leveraging AgentSpeak for the logic layer, MQTT and HTTP for asynchronous communication, the system ensures that each agent can update its beliefs and intentions dynamically with Digital Twins, which are the sensing elements and the bridge that connects agents with the physical entities. This architecture allows the Smart Road to transition from a static environment to a responsive, self-organizing ecosystem.

## 4.1 Architecture

The overall architecture is inspired by the Agents & Artifacts model and by principles of Intelligent Pervasive Systems. In this perspective, Digital Twins can be seen as artifacts that mediate the interaction between agents and the environment, providing observable properties and executable operations.
This design promotes modularity, scalability, and a clear abstraction of the physical world, enabling the development of adaptive and context-aware systems, as we can see in the figure fig. 1.

The final architecture of the system involves multiple technologies that cooperate as one single entity. The layered stack starts from the physical emulation of the real world, then the digital shadowing layer, and finally the agents layer, with bidirectional communication channels between subsequent layers. The car agent directly acts on the physical world (through the car action) but gets the information about the other car and the traffic light states via the digital twins fig. 2. We also use the concept of edge agents that are agents directly running on the physical object (like IoT) in order to have continues and immediate perception and control on the environment (no network delay or problems).

### 4.1.1 Traffic lights

Each color change of a traffic light made by the agent is propagated through the Digital Twin into the physical asset, using specific events: every state change of the junction agent generate a change on the digital model of each traffic light and that change is

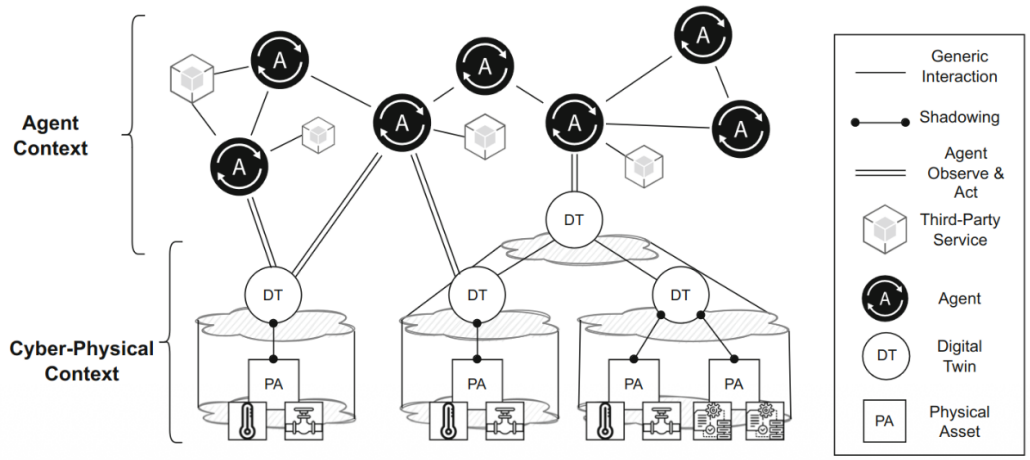


**Fig. 1** *Separation of concerns*: DTs operate within the boundaries set by the *local context* of the associated PA, whereas agents operate within the boundaries of the *global context* of the whole application set by the application designer

Figure 1: CP-MAS high-level architecture

propagated into the physical asset through appropriate adapters, so we can see the real effects of the agents behavior right inside the physical traffic light, or its emulation.

This architecture allows us to develop scalable and layered systems instead of a monolith, and separate the decision layer from the digital shadowing and the real asset. Basically, we have bidirectional communication between agents and Digital Twins: events and actions. Events refer to changes occurred on the PA (Physical Asset) which are propagated to the agent through its corresponding DT, usually using event-based technologies such as MQTT. Actions are changes decided by the agent that are propagated to the PA through its corresponding DT. Junction agent:

- Light change event: it refers to the DT's color light physical property change event, which is propagated from the bottom physical layer to the junction agent, allowing the continuous perception of the current state of the real asset
- Change light action: it is the change light action decided by the agent following its schedule policy (for example, the temporization of each color) that is propagated down to the PA using HTTP requests through the DT

Timer agent:

- Update timer action: it is an HTTP action that changes the remaining time property of the traffic light DT

Each junction agent has a timer agent with which to communicate: this allows a better modular architecture, having independent timers for each junction. We can see a simple abstraction of this concept in the following image fig. 3.

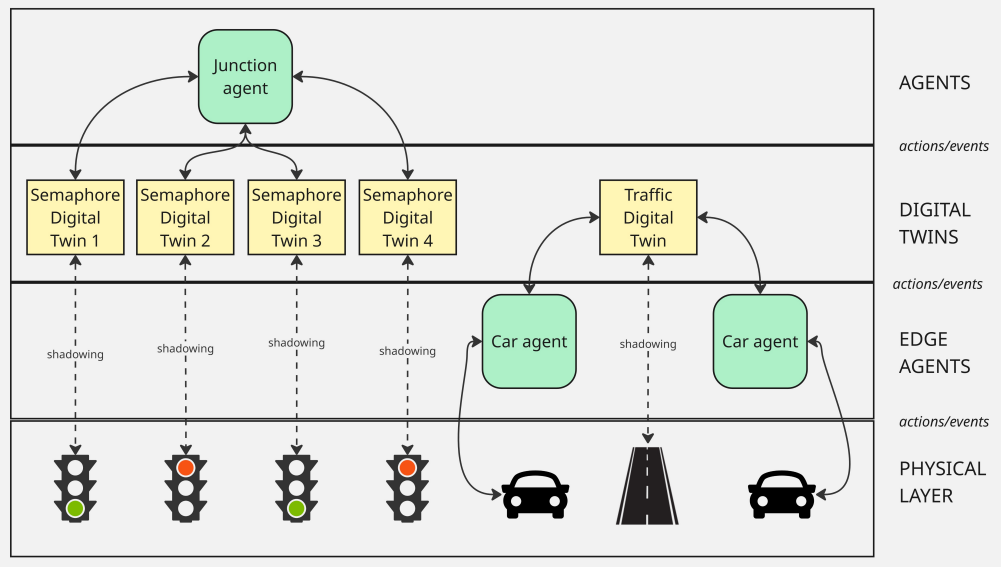


Figure 2: High-level architecture composed of agents and digital twins

Next, we can distinguish another type of communication: the agent-to-agent messages. We need the agents to cooperate to perform the real-world behavior we intend. Now let's see a more detailed view of the agents' Digital Twins interactions in the image fig. 4.

### 4.1.2 Car

The car is the only real physical element on the street, while traffic lights and signs could only be digital (could not have a real space or equivalent in the real world). The cars must move on the road to reach a given destination following a given path (from point A to point B). Since the car's agent my need to make life-or-death decision and need to monitor the situation constantly they are on board and not on the cloud, like for the junction, as we can see in the fig. 3. This is done in order do avoid connection problem (may lose signal) or network delay. They only perceive the world through the sensor on the car, the requests to the service, and the message from the traffic digital twin, which is used to map the position to the other elements. In this project, since the whole scene is simulated, the car does not have a real sensor, but it only receives the information via the broker (dt).

## 4.2 Model

### 4.2.1 Road

The road network in this project is composed of multiple roads, each represented as a list of traffic flows (directions). This representation allows the same physical road to be used for both directions of travel when required.

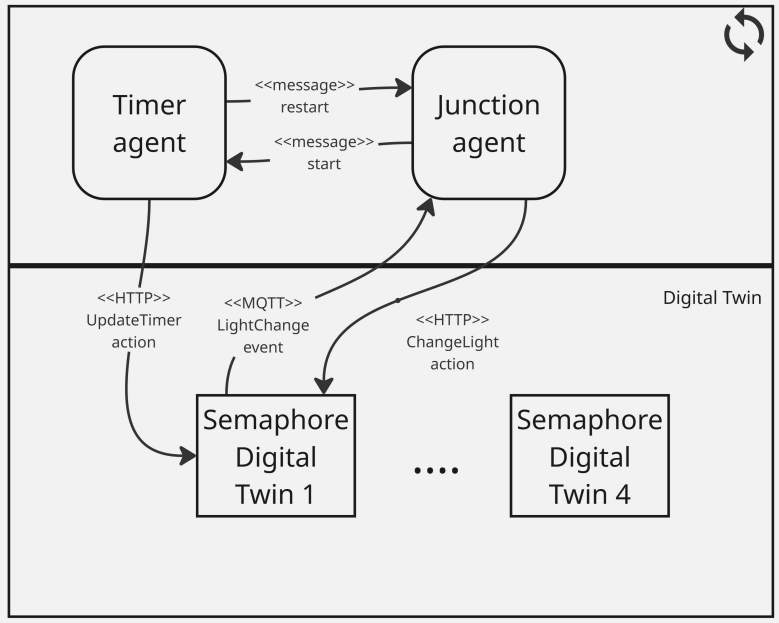


Figure 3: Agents and DTs communication

Each flow is simplified as a polyline (Section 2). The order of the segments is implemented as a sequence of points (2D), where each point creates a segment with the previous and the next, the sequence also defines the direction that vehicles must follow. A flow may contain multiple lanes; however, all lanes must be represented by segments parallel to the ones in the polyline defined by the list of points (all the streets in the scene use 1 lane).
It is worth noting that increasing the density of the flow points allows real roads to be digitized with greater accuracy.

These modeling choices avoid enforcing a right-hand traffic rule and allow the system to adapt to different scenarios, such as one-way roads or roundabouts (not implemented, but considered during the model design).

### 4.2.2 Traffic signs

Each traffic sign is described by the flow where it is, its position (2D point), and its type; the latter has the meaning of the sign (stop, yield, speed limit). For the speed limit sign, there is also the speed limit as a value; this distinction is based on the fact that all speed limit signs have the same meaning, but they only change the speed that must be enforced.

### 4.2.3 Junction

The junction is simply a connection of multiple flows and a center point. The traffic lights have different positions and are connected to the flow that they control, like the

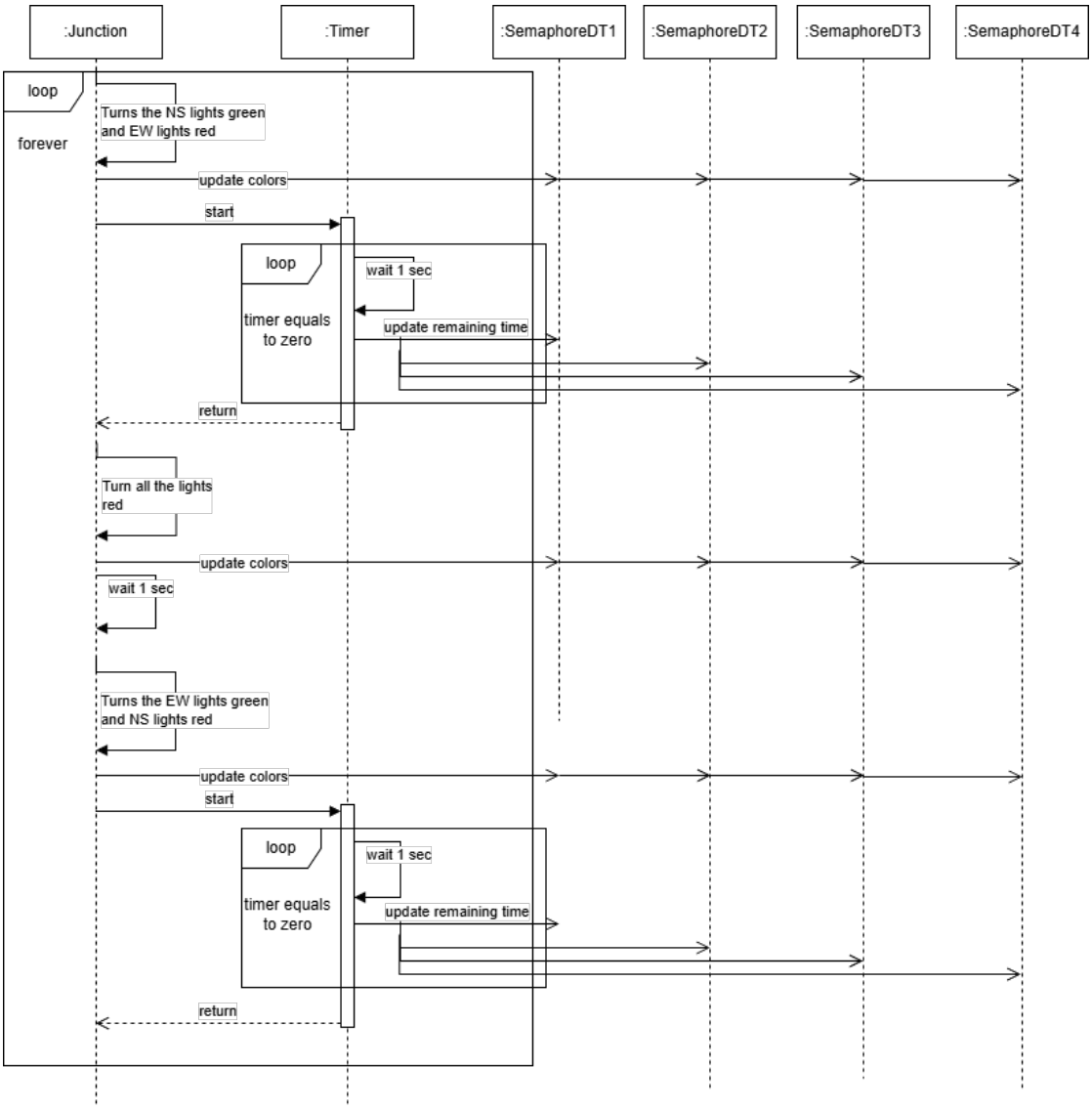


Figure 4: Activity diagram of the junction

signs. Traffic lights also have the color of the light in their state and the time remaining before the next color change.

### 4.2.4 Traffic

To calculate the car ahead in traffic, the cars' positions are put in their flow in order, for each lane. This allows us to identify the correct order of the cars.

### 4.2.5 Car

The model of the car is composed of its position and its speed. Each car requests a Path from a start to a destination, both of which are points. The path is the sequence of flows required to reach the destination. Each part of the path describes the flow, from which point to which point the car must utilize it.

### 4.2.6 Optimization

Since a car must know its position in the flow (which point it has passed), an optimization has been made for the traffic. In the flow, each segment is used as a block; the car, based on its position, searches its own block and knows the point to reach next. Since the sequences are shared by all components (car, server, traffic digital twin), it's possible to update only the index of the block (or point passed), used as a reference, and the distance (calculated as a vector) of the car from that point. With those is possible to use a relative position of the car from the start point in the block and the search for the car ahead (in the traffic) with a simple ordered array.

### 4.2.7 Improvements to do

An important optimization that only emerged at the end of the project is that the logic used for the cars in the traffic could be used for both signs and traffic lights. That would have made a much more heavy computation, especially the part where the data are saved in the system, but a more simplified calculation for both agent and simulation (more common operation).

### 4.2.8 How the model is divided

The described model is the one that emerges from the whole system. Each component of the system only has the parts of the model that are needed to be able to reach its goal.

- The simulation doesn't know the entire system, but only its surface knowledge
- The car's agent has deep knowledge of the entire system.
- The microservices and Digital Twins have deep knowledge of the part of the system that has been assigned to each of them.

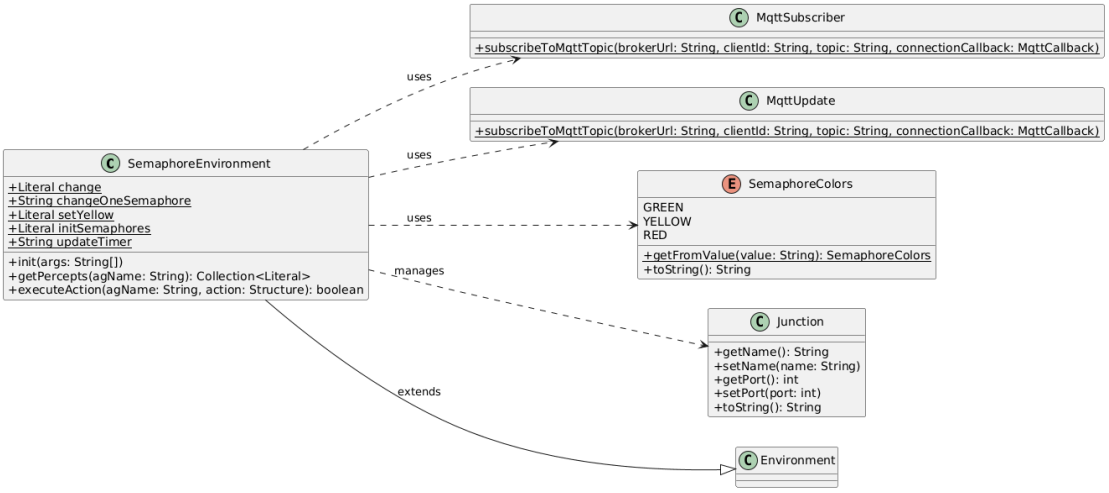


Figure 5: Structure of the Junction MAS

We can see a more detailed structure in the following UML diagrams fig. 5 and fig. 6.

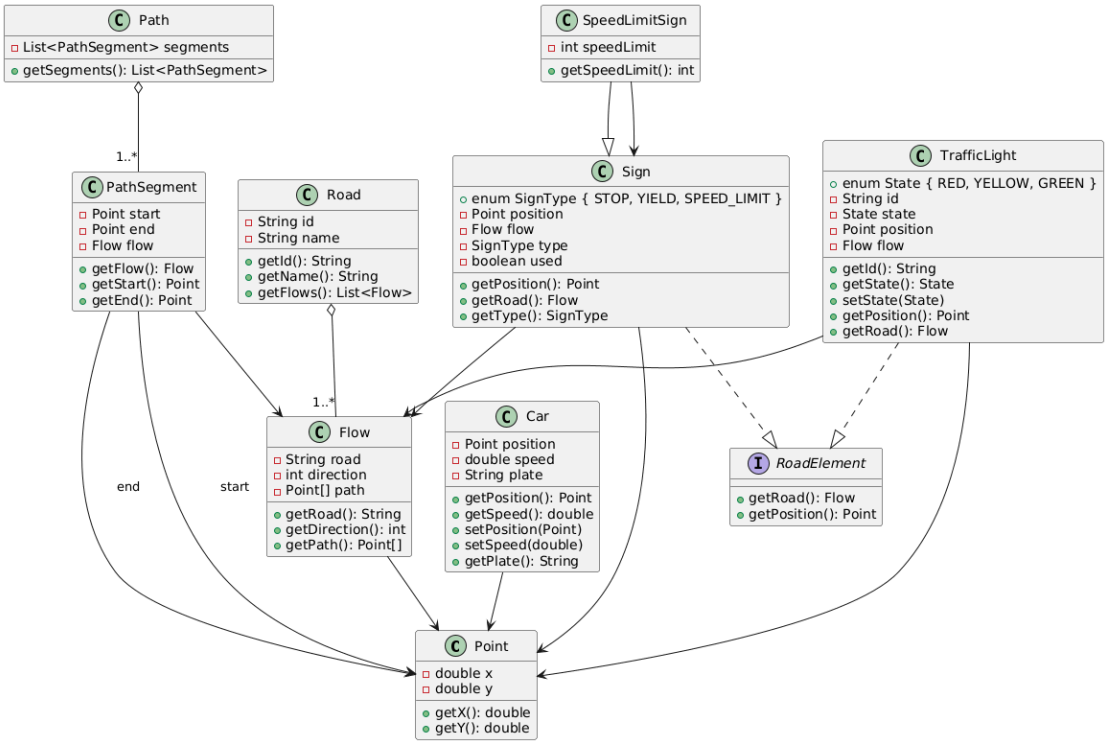


Figure 6: Model of the whole system

## 4.3 Car Multi Agent System

The main part of this system is defined by the Car Agent. Each car has an agent their behaviors, so there isn't a central intelligence but the end result is the union of "simple" single agents

The car agent is the one responsible to control the car on the road and decide if the car should stop, speed up, or keep the distance. The movement part of the agent is developed in Java since it is extremely mathematically complex and hardly fits with the idea of PROLOG's rules or the AgentSpeak BDI (Beliefs-Desires-Intentions) structure.

### 4.3.1 Interaction and behavior

The single car agents do not interact with each other or other agents directly (like the junction agent), but use the object of the street to communicate (the traffic Digital Twin and the Semaphore Digital Twin). As a first thing, the car agent must know its starting point, its destination, and its path on the map.

1. Retrieves the path from the services (the paths are pre-configured at this point and are assigned on the order of arrival of the request, the first agent gets the first path, and so on cyclically once the last is given).
2. Retrieves the flow used in the assigned path, also from the service
3. Retrieves the traffic sign on the flows of the path
4. Retrieves the traffic lights on the flows of the path
5. Subscribes to the traffic lights on the flows of the path
6. Subscribes to get the car ahead in traffic (the car could change during the path)

Then, when the engine of the car starts (user action), the car has 2 goals:

- Reaching the destination
- Drive safely on the roads if it is in autonomous driving

The first can be replaced with following the given pre-calculated path. The second is the most interesting and consists of keeping a speed at which the car can safely move on the street. To do so, the car must:

- Keep a speed below the speed limit (given by the last speed limit sign)
- Keep a safe distance (based on its speed) from the car ahead
- Stop at Stop signs and red traffic lights

A rule, added by us, is that the car should prefer the gentle slowing down (without using the brakes) to using the brakes. This is done to be more fuel efficient (for both petrol and electric cars).

This gives the agent 4 states :

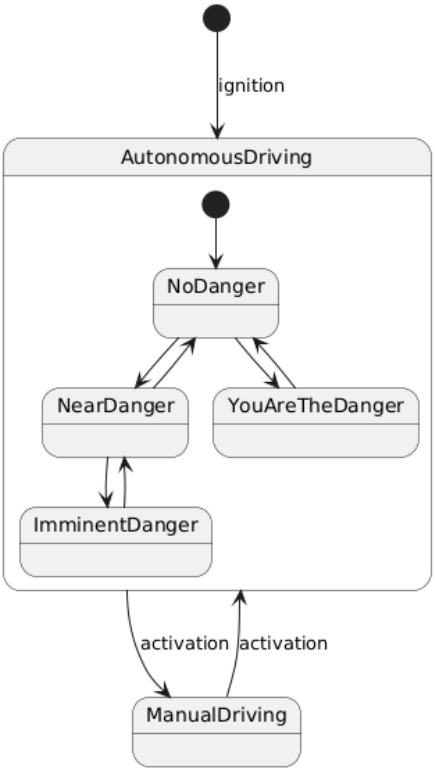


Figure 7: State diagram of the car agent for the speed

- Immediate danger
- Near danger
- No danger
- You are the danger

The transaction can be seen in the diagram state (fig. 7) The transactions aren't explicit in code, but it is rational to think that when there is a danger, you first see it near and then closer. The only exception is when the traffic lights turns from green to yellow (the rule is to stop). The "You are the danger" state is applied only when the agent is too slow, so it must speed up.

## 4.4 Junction Multi Agent System

A core component of the Smart Road design is the specialized Multi-Agent System dedicated to intersection management. This sub-system is engineered to handle the complexity of urban crossings through a coordinated effort between two different types of specialized agents:

- Junction Agent: This agent acts as the local orchestrator for a specific crossing. Manages a cluster of four traffic lights that are modeled as Digital Twins. These digital replicas allow the agent to monitor the state of each semaphore in real-time, ensuring maximum reliability and synchronization
- Timer Agent: Each intersection is assigned a dedicated Timer Agent. This agent is responsible for the temporal logic and to allow execution of different specific traffic policies for future works (e.g., fixed cycles, adaptive timing based on congestion, or emergency overrides). By decoupling the timing logic from the intersection's structural management, the system gains flexibility in adjusting policies dynamically without affecting the underlying infrastructure control

This hierarchical approach ensures that each intersection can operate autonomously while remaining open to global optimizations received from the broader Smart Road network.

### 4.4.1 Interaction and behavior

Interactions between agents and the environment are the core point of an agent system-based architecture. We have the intelligent agents on one side, which represents the brain of the entire system, and the digital twins on the other side, representing the digital shadowing of the physical entities.

Now let's have a look inside the junction agent: the image fig. 8 represents, from a high level of abstraction, the aggregation of 4 different traffic lights, represented in the image as North-East-South-West.

1. We start at the very first state with two red and two green traffic lights
2. Then, after a predefined amount of time, it changes to the next state, the yellow phase, where the cars that cannot stop securely are allowed to proceed
3. Next to each yellow phase, we always have a so-called security phase: all the traffic lights turn red for one second, to prevent crashes for vehicles that fail to stop during the yellow phase
4. Then we start a new cycle but with the opposite initial state, and the two red lights become green, and the previous green lights (then turned yellow) remain red
5. Again, the yellow phase after the predefined amount of time
6. Always the security phase for 1 second
7. Finally repeat the cycle from point 1

The timer agent is much simpler: its behavior consists of waiting for a timer request from the junction agent and starting the time countdown until it reaches zero. When the timer ends, a restart message to the junction agent is sent in order to change the colors of the traffic lights and continue the semaphore cycle. A basic state diagram is shown in the following image fig. 9.


Figure 8: High-level state diagram of traffic light

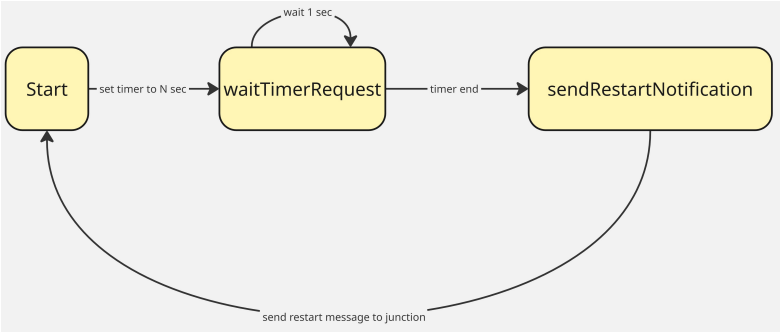


Figure 9: State diagram of the timer agent

### 4.5 Corner cases

On the car agent, when none of the defined plans can be applied, in order to avoid creating crashes with other cars, the car's plan is to slow down. This shouldn't normally happen, but it is a fail-safe designed to handle unexpected scenarios.

## 5 Salient implementation details

This section details the technical implementation of the system, focusing on the Multi-Agent Systems' reasoning and sensing ( done using Digital Twin technology).

The implemented architecture (as mentioned before) follows a layered architecture designed to decouple data acquisition from cognitive logic:

- Physical Layer: Comprises the sensors and actuators embedded in the physical asset.
- Edge Agent Layer: Autonomous entities that directly acts on the physical system.
- Digital Twin Layer: A high-fidelity virtual representation that mirrors the state of the physical system using real-time data streams.
- Agent Layer: Autonomous entities that monitor the Digital Twin, perform reasoning, and execute decision-making processes.

### 5.1 Agent Logic and Interaction

Agents are implemented using a BDI model (Belief-Desire-Intention). Each agent is responsible for a specific subset of the Digital Twin's parameters:

- Monitoring Agents: Constantly listen to DT events for state changes to perceive the physical world and correct anomalies.
- Domain Logic: Execute the intended plans to perform the expected behavior and domain logic.
- Control Agents: Translate high-level decisions into actionable commands sent back to the physical asset.

### 5.2 Timer Agent

The **Timer Agent** is a specialized component designed to handle temporal constraints and state transitions for the traffic infrastructure. Its operational logic follows a reactive communication protocol with the *Junction Agent*. The Timer agent waits for a start request from the Junction agent. Once the request is received, the Timer agent begins its countdown. After the specified time period has elapsed, the Timer agent sends a message back to the Junction agent to indicate that the timer has finished, prompting a state change in traffic lights.

Listing 1: Timer agent behavior

```
time(5000).

+!start : true <-
    !wait_timer_request;
    !send_restart_notification;
    !start.

+!wait_timer_request: other(Sender) <-
    .wait({ +start[source(Sender)] });
    -start[source(Sender)];
    !waitSem.

+!waitSem <-
     while(time(X) & X > 0) {
         !!updateTimerPlan(X);
         .wait(1000);
         -+time(X - 1000);
     };
     -+time(5000).
```

## 5.3 Junction agent

The junction agent manages four traffic lights, operating in a cycle of four phases described above. In the first phase, two lights are red while the other two are green. In the second phase, the two green lights turn red, and the two red lights turn green. Between these two phases, we have the yellow lights and the secure phase of red lights. The agent coordinates the lights in this sequence to ensure the safe and efficient flow of traffic.

Listing 2: Junction agent behavior

```
+!temporalPlan : semaphore(1, green) &  semaphore(2, red) & semaphore(3, green) &
    semaphore(4, red) <-
    !send_request;
    !wait_response;
    !yellowBehavior.

+!temporalPlan : semaphore(1, red) &  semaphore(2, green) & semaphore(3, red) &
    semaphore(4, green) <-
    !send_request;
    !wait_response;
    !yellowBehavior.

+!temporalPlan : semaphore(1, red) &  semaphore(2, yellow) & semaphore(3, red) &
    semaphore(4, yellow) <-
    !send_request;
    !wait_response;
    changeOneSemaphore(red, 2, 4);
    .wait(1000);
    changeOneSemaphore(green, 1, 3);
    !temporalPlan.

+!temporalPlan : semaphore(1, yellow) &  semaphore(2, red) & semaphore(3, yellow)
    & semaphore(4, red) <-
   !send_request;
   !wait_response;
   changeOneSemaphore(red, 1, 3);
   .wait(1000);
```

```
24      changeOneSemaphore(green, 2, 4);
25      !temporalPlan.
```

The temporal plan is based on the agent's actual beliefs, which depend on the actual color of the traffic lights. We can note the action made by the agent in order to change the effective traffic light.

### 5.3.1 Data Synchronization and Communication

The synchronization between the Digital Twin and the agents is facilitated through a robust communication middleware (e.g., MQTT and HTTP).

- Ingestion: Real-time telemetry is pushed from the physical sensors to the Digital Twin.
- Shadowing: The virtual model updates its state variables to reflect the physical reality.
- Analysis: Agents query the updated model and compare the current state against predefined performance thresholds.
- Feedback Loop: If an intervention is required, the agent triggers an action through the Digital Twin's actuation interface, which then commands the physical hardware.

The junction agent sends POST requests to the Digital Twins of the traffic lights to change their colors according to the current phase. When it's time to switch the lights, the agent communicates with the Digital Twins by sending the appropriate requests to change the state of each light (e.g., red, yellow, or green). These requests ensure that the traffic lights update their colors in sync with the agent's programmed schedule, maintaining the flow of traffic at the intersection. The *executePost* method is used to send an update to the corresponding traffic light Digital Twin, sending a simple HTTP request with the color information, while the *updateDtTimer* is used to update each traffic light timer property via MQTT dedicated channels. We combine multiple technologies to have a complete interaction and integration between the agent and DT layers.

Listing 3: Junction environment

```
1  private void executePost(String color, String agName, int id) throws IOException,
       InterruptedException {
2      HttpClient client = HttpClient.newBuilder().build();
3      String json = "{\"color\":\"" + color + "\"}";
4      int port = junctionBasePorts.get(agName) + id;
5      String url =  "http://" + dtHost.getValue() + ":" + port + targetUrl;
6      HttpRequest request = HttpRequest.newBuilder()
7              .uri(URI.create(url))
8              .header("Content-Type", "application/json")
9              .POST(HttpRequest.BodyPublishers.ofString(json))
10             .build();
11     HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers
           .ofString());
```

```
}

private void updateDtTimer(int secondsRemaining, String agName) throws IOException
    , InterruptedException {
    MqttUpdate mqttUpdate = new MqttUpdate(mqttHost.getValue(), mqttPort.getValue
        (), "semaphore-agent");
    Stream.iterate(0, i -> i + 1).limit(4).forEach( i -> {
        mqttUpdate.publishUpdate("semaphore/" + agName + "_" + i + "/
            remaining_time", String.format("%d", secondsRemaining));
    });
}
```

### 5.3.2 Perception

In Jason, the perception mechanism plays a crucial role in enabling agents to maintain an up-to-date and coherent view of the environment in which they operate. Perceptions are continuously gathered from the MAS environment and used to update the agent's belief base, forming the foundation for all subsequent reasoning and decision-making processes. In the case of the junction agent, for example, its beliefs are updated with the current state of all the traffic lights within the junction. This allows the agent to react dynamically to environmental changes, captured by Digital Twins, and to coordinate its actions based on the latest available information. For example, the following code snippet represents the perception update for the junction traffic lights: all *semaphoreStates* of all agents are updated by listening for every color change event of the traffic lights in the Digital Twin, to react in case of unexpected changes.

Listing 4: Junction perception

```
@Override
public Collection<Literal> getPercepts(String agName) {
    final Set<Literal> set = new HashSet<>();
    String index = agName.replaceAll("\\D", "");
    if(agName.contains("timer")) {
        Literal literal = Literal.parseLiteral("other(junction" + index + ")");
        set.add(literal);
    } else {
        Literal literal = Literal.parseLiteral("other(timer" + index + ")");
        set.add(literal);
    }
    if(semaphoresStates.containsKey(agName)) {
        Literal semaphore1 = Literal.parseLiteral(String.format("semaphore(%s,%s)"
            , "1", semaphoresStates.get(agName).get(0)));
        Literal semaphore2 = Literal.parseLiteral(String.format("semaphore(%s,%s)"
            , "2", semaphoresStates.get(agName).get(1)));
        Literal semaphore3 = Literal.parseLiteral(String.format("semaphore(%s,%s)"
            , "3", semaphoresStates.get(agName).get(2)));
        Literal semaphore4 = Literal.parseLiteral(String.format("semaphore(%s,%s)"
            , "4", semaphoresStates.get(agName).get(3)));
        set.add(semaphore1);
        set.add(semaphore2);
        set.add(semaphore3);
        set.add(semaphore4);
    }

    return set;
}
```

## 5.4 Car Agent

The Car agent defines the rules for the autonomous driving strategy.

### 5.4.1 Belief

The car agent's beliefs are the following:

Listing 5: Car agent's beliefs

```
stop.
red.
yellow.
green.

driver.
autonomous.
mode(driver).


car_on.
car_off.

car_state(car_off).

break_speed(5).
drag_speed(0.5).
reaction_time(1).
car_distance(70).
```

These represent:

- The possible color of the traffic lights
- The state of the autonomous drive
- The state of the car
- The given (simulated) value for break and slowing down
- The required safe distance from the car (point to point)

### 5.4.2 Rules

To calculate the danger (or hazard), the car needs a position rule based on the element on the road.

Listing 6: Car agent's math rules

```
near(X1, Y1, X2, Y2, DN) :-
    distance(X1, Y1, X2, Y2, D) & D < DN.

distance_to_stop(CS, DECEL, SD) :-
    reaction_time(RT)
    & SD = (CS * CS) / (2 * DECEL) + (CS * RT).

stopping_distance(CS, SD) :- break_speed(BS) & distance_to_stop(CS, BS, SD).
```

```
dragging_distance(CS, SD) :- drag_speed(DS) & distance_to_stop(CS, DS, SD).

stopping(X1, Y1, X2, Y2, CS) :- distance(X1, Y1, X2, Y2, D)
                                & stopping_distance(CS, SD)
                                & D < SD + 10.

dragging(X1, Y1, X2, Y2, CS) :- distance(X1, Y1, X2, Y2, D)
                                 & dragging_distance(CS, SD)
                                 & D < SD + 10.

distance(X1, Y1, X2, Y2, D) :-
    utils.sqrt(D, (X1 - X2) * (X1 - X2) + (Y1 - Y2)*(Y1 - Y2)).

/* Safe distance rules */

/* Normal safe distance: proportional to front car speed */
normal_safe_distance(S, SD) :- car_distance(D) & SD = S / 2 + 10 + D.

/* Dynamic safe distance: based on relative speed (if faster than front car) */
dynamic_safe_distance(CS, OS, SD) :-
    car_distance(D) &
    Diff = CS - OS &
    Diff > 0 &
    SD = D + 10 + Diff * 2.

dynamic_safe_distance(CS, OS, SD) :-
    Diff = CS - OS &
    Diff <= 0 &
    normal_safe_distance(S, SD).
```

These rules are general knowledge for drives and are used to calculate:

- The concept of near used to define when an object is close
- The distance needed to stop (for both dragging, no brakes, and brakes)
- The safe distance from a car ahead (both faster or slower)

The second set of rules are needed to calculate the danger of the elements (information) on the road.

Listing 7: Car agent's danger rules

```
/* Define which elements require the car to stop or slow down */
arrest_car_event(E, SX, SY) :- element(stop, SX, SY) & E = stop.
arrest_car_event(E, SX, SY) :- element(traffic_light(red), SX, SY) & E =
    traffic_light(red).
arrest_car_event(E, SX, SY) :- element(traffic_light(yellow), SX, SY) & E =
    traffic_light(yellow).

/* ------------------------------------------------
   Hazard levels
--------------------------------------------------*/

hazard(Level, E) :-
    arrest_car_event(E, SX, SY)
    & currentSpeed(CS)
    & position(X, Y)
    & stopping(X, Y, SX, SY, CS)
    & CS > 0
```

```
    & Level = 2.

hazard(Level, E) :-
    car(D, OS)
    & position(CX, CY)
    & currentSpeed(CS)
    & CS > OS
    & dynamic_safe_distance(CS, OS, DSD)
    & D < DSD
    & Level = 2
    & E = car(D, OS).

hazard(Level, E) :-
    arrest_car_event(E, SX, SY)
    & currentSpeed(CS)
    & position(X, Y)
    & dragging(X, Y, SX, SY, CS)
    & CS > 15
    & Level = 1.

hazard(Level, E) :-
    car(D, S)
    & position(CX, CY)
    & currentSpeed(CS)
    & dynamic_safe_distance(CS, S, DSD)
    & normal_safe_distance(S, NSD)
    & D >= DSD
    & D <= NSD
    & Level = 1
    & E = car(D, S).

hazard(Level, E) :-
    arrest_car_event(E, SX, SY)
    & position(X, Y)
    & near(X, Y, SX, SY, 10)
    & currentSpeed(0)
    & Level = 0.

hazard(Level, E) :-
    car(CD, S)
    & car_distance(D)
    & currentSpeed(0)
    & CD < D
    & Level = 0
    & E = car(CD, S).

/* Determine hazard level based on current speed and speed limit */

hazard(Level, E) :-
    currentSpeed(CS)
    & speedLimit(SL)
    & CS > SL + 20
    & Level = 2
    & E = speed_limit_violation.

hazard(Level, E) :-
    currentSpeed(CS)
    & speedLimit(SL)
    & CS > SL
    & CS <= SL + 20
    & Level = 1
    & E = speed_limit_violation.
```

```
78
79  hazard(Level, E) :-
80      currentSpeed(CS)
81      & speedLimit(SL)
82      & CS <= SL
83      & CS >= SL - 5
84      & Level = 0
85      & E = no_hazard.
86
87  hazard(Level, E) :-
88      currentSpeed(CS)
89      & speedLimit(SL)
90      & CS < SL
91      & Level = -1
92      & E = less_hazard.
```

These rules define which elements are dangerous and at what level they pose, based on their distance (calculated using the position). To handle the speed, the level of danger makes the agent decides which action must be taken (brake, slow down, maintain speed, or speed up).

### 5.4.3 Behavior

At this point, the behavior is nothing more than defining what action must be done based on the level of danger.

Listing 8: Car agent's behavior

```
1
2   +!reachSpeedLimit : hazard(2, E) <-
3       .print("Moderate hazard detected, braking");
4       .print("Hazard event: ", E);
5       brake;
6       .wait(1000);
7       !reachSpeedLimit.
8
9   +!reachSpeedLimit : hazard(1, E) <-
10      .print("Minor hazard detected, slowing down slightly");
11      .print("Hazard event: ", E);
12      do_nothing;
13      .wait(1000);
14      !reachSpeedLimit.
15
16  +!reachSpeedLimit : hazard(0, E) <-
17      .print("No hazards detected, keeping speed");
18      .print("Hazard event: ", E);
19      keep_speed;
20      .wait(1000);
21      !reachSpeedLimit.
22
23  +!reachSpeedLimit : hazard(-1, E) <-
24      .print("No hazards detected, proceeding to reach speed limit");
25      .print("Hazard event: ", E);
26      accelerate;
27      .wait(1000);
28      !reachSpeedLimit.
29
30  +!reachSpeedLimit <-
```

```
    .print("No plans applicable, slowing down to be safe and avoid potential
        problems");
    do_nothing;
    .wait(1000);
    !reachSpeedLimit.
```

### 5.4.4 Perception

Each time the car moves, the previous perceptions of position, speed, speed limit, and road elements ahead are removed and re-added (update).

Listing 9: Car agent's perception update

```
private void updatePercepts() {
        updatePositionPercept();
        updateSpeedPercept();
        var speedLimit = updateSpeedLimitPercept();
        var signals = updateSignalsPercepts();
    }

    private void updatePositionPercept() {
        removePerceptsByUnif(Literal.parseLiteral(POSITION_BASE));
        addPercept(positionToLiteral(car.getPosition()));
    }

    private void updateSpeedPercept() {
        double scaleSpeed = scaler.descaleValue(car.getSpeed());
        removePerceptsByUnif(Literal.parseLiteral(CURRENT_SPEED_BASE));
        addPercept(currentSpeedToLiteral(scaleSpeed));
    }

    private String updateSpeedLimitPercept() {
        removePerceptsByUnif(Literal.parseLiteral(SPEED_LIMIT_BASE));
        return addSpeedLimitPercept();
    }

    private String addSpeedLimitPercept() {
        int speedLimit =
                RoadElements.getLastSpeedLimitSign(
                                roadsElementsVision.getPassedRoadElements()
                        ).map(SpeedLimitSign::getSpeedLimit)
                        .orElse(defaultSpeedLimit);
        var literal = speedLimitToLiteral(speedLimit);
        addPercept(literal);
    }

    private String updateSignalsPercepts() {

        removePerceptsByUnif(signTypeToLiteral(Sign.SignType.STOP));

        var literalStop = RoadElements.getNextStopSign(
                roadsElementsVision.getNextRoadElements()
        ).map(LiteralConverter::signToLiteral);
        literalStop.ifPresent(this::addPercept);

        removePerceptsByUnif(Literal.parseLiteral(TRAFFIC_LIGHT_BASE));
        var literalTrafficLight =
                RoadElements.getNextTrafficLight(
                        roadsElementsVision.getNextRoadElements()
                ).map(LiteralConverter::trafficLightToLiteral);
```

```
48          literalTrafficLight.ifPresent(this::addPercept);
49      }
50
51      private void updateOtherCarPercept(Optional<OtherCar> car) {
52          removePerceptsByUnif(Literal.parseLiteral(OTHER_CAR_BASE));
53          if(car.isEmpty()) {
54              otherCar = Optional.empty();
55              updatePercepts();
56          }else {
57              addPercept(otherCarToLiteral(car.get(), scaler));
58              otherCar = car;
59              updatePercepts();
60          }
61      }
```

The other car perceptions and the traffic light perception (actually, the whole signs ahead to be precise) are updated based on the message received by the traffic digital twin and the semaphore digital twin, respectively.

# 6 Deployment Instructions

We provide one simple scenario with two roads consisting of a single lane each, and one junctions with four traffic lights.

## 6.1 First step: run the Docker containers

First of all, we need to run the Docker containers, such as microservices for roads and signals storage, Digital Twin infrastructure, and communication services such as Mosquitto. Run the following command inside the *RunningSystem* repository:
*docker compose up*

## 6.2 Second step: execute the scenario

Now we need to execute the scenario to populate all the microservices' databases with starting information and create all the necessary Digital Twins. Run the following command inside the *RunningSystem* repository:
*node scene.js*

## 6.3 Third step: execute the MAS

Now that we have all the containers running, we need to execute the cars and junctions MAS as a normal application: in this way, we are able to better analyze the agents running correctly. To create a new single car agent run the following command inside the CarMAS repository folder:
*./gradlew runcarMAS*
To create the agent of the traffic light of the scenario run the following command inside the JunctionMAS repository folder:
*./gradlew runjunctionsMAS*

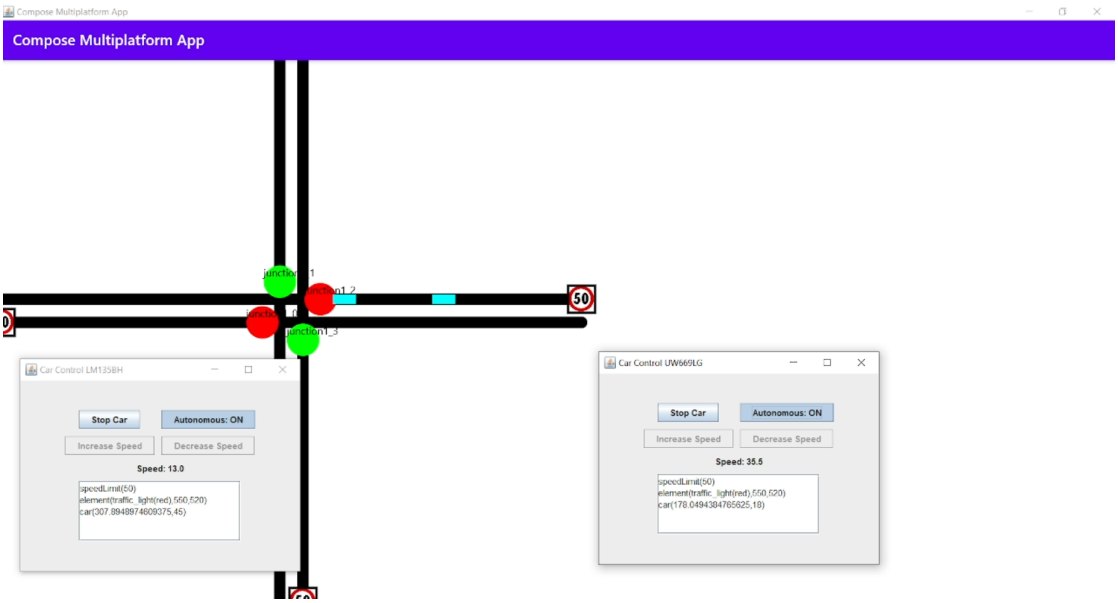


Figure 10: Simulation

# 7 Usage Examples

Let's explain the working rules after having executed all the necessary components: firstly we have to start each car with autonomous drive, clicking on *Start Car* (the license number is not necessary) and then click to *Autonomous: OFF* to start the autonomous driving (it will appear the ON label). The referring image is fig. 10.

# 8 Conclusions

In conclusion, combining agent-oriented programming with digital-twin technology within a single project has proven to be an extremely valuable and formative experience. These two paradigms represent key technological directions for the future, and their integration — even in the form of a simple experimental prototype — highlights their potential when applied to autonomous driving scenarios. Agent-based systems provide a natural way to model autonomous, goal-directed behaviour, while Digital Twins enable continuous interaction with a realistic virtual representation of the environment. Working at the intersection of these technologies has offered meaningful insight into how intelligent, adaptive, and context-aware systems may be developed and tested safely before being deployed in the real world.

## 8.1 Future Works

As said in the model section, the first improvement would be change the model to avoid calculating for each car the position of the road element based on the block in which the

car is.

The second improvement would be to integrate the concept of remaining time in the car agent so that the action near the traffic light could be more precise, and augment the reality (only a few traffic lights show the remaining time of the green or yellow, and that could help drivers, and autonomous drivers alike, to make a decision when approaching them).