Decentralized configuration among autonomous vehicles via cognitive stigmergy MAS

1.0 Introduction

This project concerns the creation of a multi-agent system that manages the coordination and modeling of autonomous vehicles. Its distinctive feature is the approach to communication and coordination between agents, which is entirely based on the concept of cognitive stigmergy. Coordination between agents does not take place through direct communication, but through the use of a shared space, in which each agent leaves its own cognitive feedback. In cognitive stigmergy, therefore, communication does not take place through simple indirect feedback, as in the case of ant swarms, but rather through the sharing of structured information that also includes the agent's intentionality.

The project was made possible through the use of JaCaMo, a framework that combines Jade, Java, and CarTaGo in a single configuration, allowing for the creation of multi-agent systems with shared environments.

The proposed features are as follows:

- creation of an environment in which agents can coexist and roads are added. Each road has two lanes with a predetermined direction to emulate real behavior;
- creation of reactive agents capable of communicating through the environment, with a stigmergic and cognitive approach; in particular, an agent's intentionality allows it to intelligently calculate its next step;
- integration of a system of timed traffic lights (cartago artifact), each with its own internal timer, which facilitates coordination;
- possibility of adding simple turns to the map;
- possibility of adding intersections to the map, which are by definition the meeting point of four different roads, and through intentionality, the mechanism governing right of way is defined;
- creation of a rather complex mechanism to randomly generate all MAS components. The generator opens up the possibility of creating potentially infinite scenarios, which can then be exploited, through metrics (currently not present), for performance calculation. The idea is to compare a controller created with cognitive stigmergy with a MAS with a classic approach and verify, through experimentation based on a large number of examples, whether there is statistical significance in the (possible) difference in performance.

2.0 Context

This section focuses on technologies and techniques needed for the project realization, it's an overview that helps to focus on MAS intent.

2.1 JaCaMo

The JaCaMo project aims to promote the Multi-Agent Oriented Programming (MAOP) approach by providing a development platform that integrates tools and languages for programming the following dimensions of Multi-Agent Systems: agents, environment and organisation. JaCaMo platform addresses applications demanding autonomy, decentralization, coordination and openness. (source: https://jacamo-lang.github.io/).
As stated and explained in the JaCaMo documentation, the platform covers three different main dimensions, such as:

- agents → autonomous entities;
- environment → shared space in which agents interact;
- organization → rules, roles and structures that facilitate coordination.

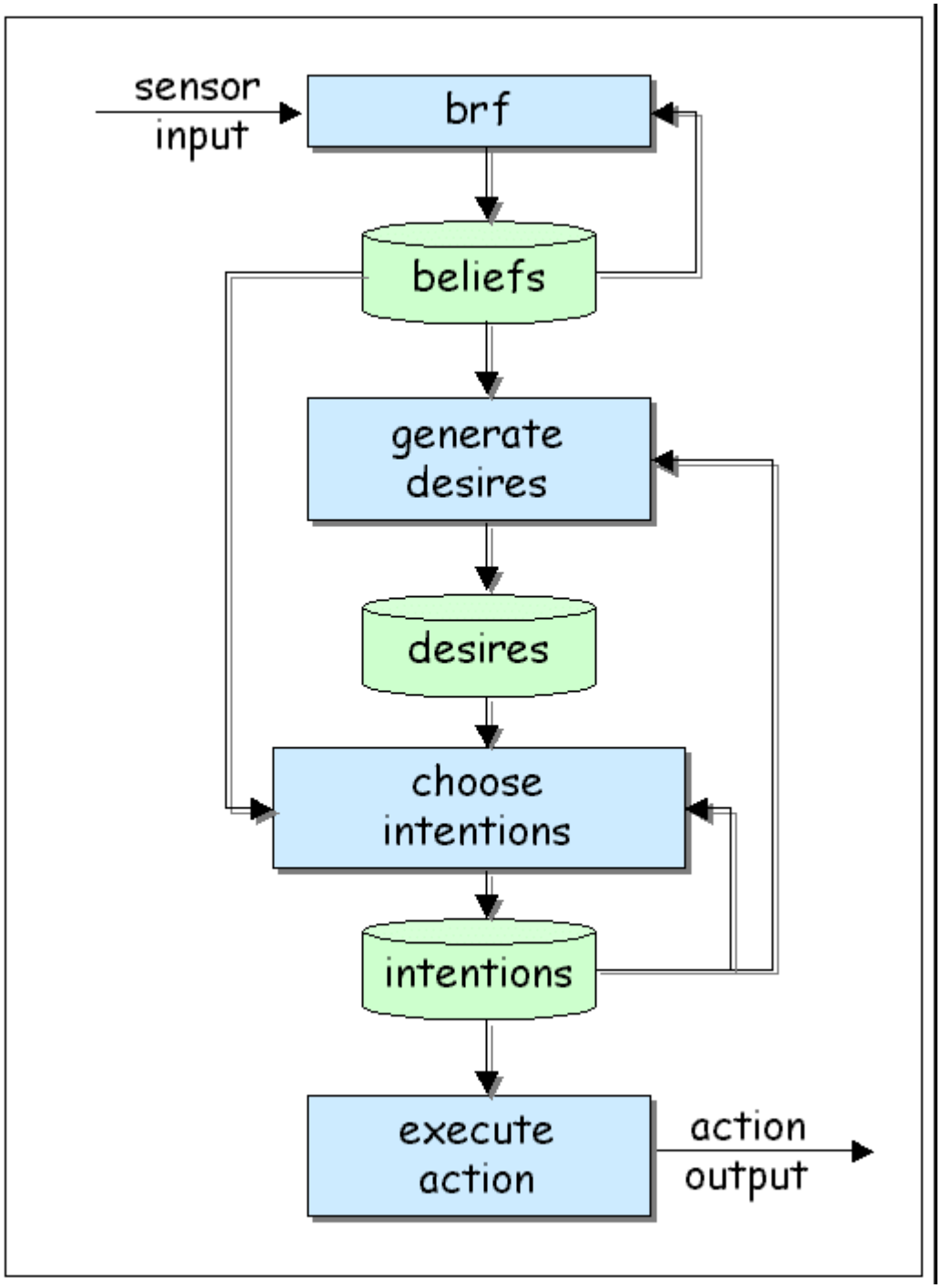
The architecture is structured in three different levels, jason, cartago and moise that can also be used separately.

2.2 Jason

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. One of the best known approaches to the development of cognitive agents is the BDI (Beliefs-Desires-Intentions) architecture. In the area of agent-oriented programming languages in particular, AgentSpeak has been one of the most influential abstract languages based on the BDI architecture. The type of agents programmed with AgentSpeak are sometimes referred to as reactive planning systems. Jason is a fully-fledged interpreter for a much improved version of AgentSpeak, including also speech-act based inter-agent communication. A Jason multi-agent system can be distributed over a network effortlessly. Various ad hoc implementations of BDI systems exist, but one important characteristic of AgentSpeak is its theoretical foundation: it is an implementation of the operational semantics, formally given to the AgentSpeak language and most of the extensions available in Jason. (source: https://jason-lang.github.io/).

BDI Architecture

As stated in the description, Jason is based on BDI architecture, since the model is very efficient for balancing a deliberative vs reactive approach. Its structure can be described by the following schema.



2.3 CArtAgO

CArtAgO is based on the Agents & Artifacts (A&A) meta-model for modelling and designing multi-agent systems. A&A introduces high-level metaphors taken from human cooperative working environments: agents as computational entities performing some kind of task/goal-oriented activity (in analogy with human workers), and artifacts as resources and tools dynamically constructed, used, manipulated by agents to support/ realise their individual and collective activities (like artifacts in human contexts). Actually, A&A is based on interdisciplinary studies involving Activity Theory and Distributed Cognition as main conceptual background frameworks. (source: https://apice.unibo.it/xwiki/bin/view/CARTAGO/).

One of Cartago main features is that it isn't passive, it's structured as a set of artifacts, organized in workspaces, such that workspaces can be distributed over the network. Cartago also offers the possibility to work with heterogeneous agents, from even different platforms, this allows a very good cooperation mechanism.

Cartago is also orthogonal, since it has a very good implementation of the separation-of-concerns mechanism, since the artifact encapsulates resources, access protocols and external services access. It's also not bounded to a specific language.

In BDI-based languages such as Jason, beliefs and events trigger plans that, in turn, execute actions. With cartago, these external actions in the BDI cycle are mapped directly to operations exposed by artifacts, while artifact properties and events are received by agents as perceptions or beliefs. This integration is natural and preserves the interpretability of the agent's deliberation cycle.

2.4 Self-organizing systems and cognitive stigmergy

This project aims to build a self-organized system, which is a process in which a pattern at the global level of a system emerges solely from numerous interactions among the lower-level components of the system. Moreover, the rules specifying interaction among the system's components are executed using only local information, without reference to the global pattern. Self organization is really common in biology.
Coordination is one of the most important aspects of coordination, in particular, there Grassé proposes an explanation for the coordination observed in termites societies "The coordination of tasks and the regulation of constructions are not directly dependent from the workers, but from constructions themselves. The worker does not direct its own work, he is driven by it. We call this particular stimulation "stigmergy."
Nowadays, stigmergy refers to a set of coordination mechanism mediated by the environment, in which agents release feedbacks indirectly, in the environment, such that they can be used by other agents in order to choose the better path or action (e.g in ant colonies chemical substances, namely pheromones, act as markers for specific activities).

Cognitive stigmergy

An interesting extension of the stigmergy concept is cognitive stigmergy that was introduced in 2007 by prof. Ricci and prof. Omicini. Agents can perceive, share and use artifacts to support individual goals without central control.
The designed system (mine), leverages the presented principle, agents coordinate indirectly via shared environmental artifacts.

3.0 Design

This chapter shows and explains all the choices made in order to build and make the system work.

3.1 Architecture

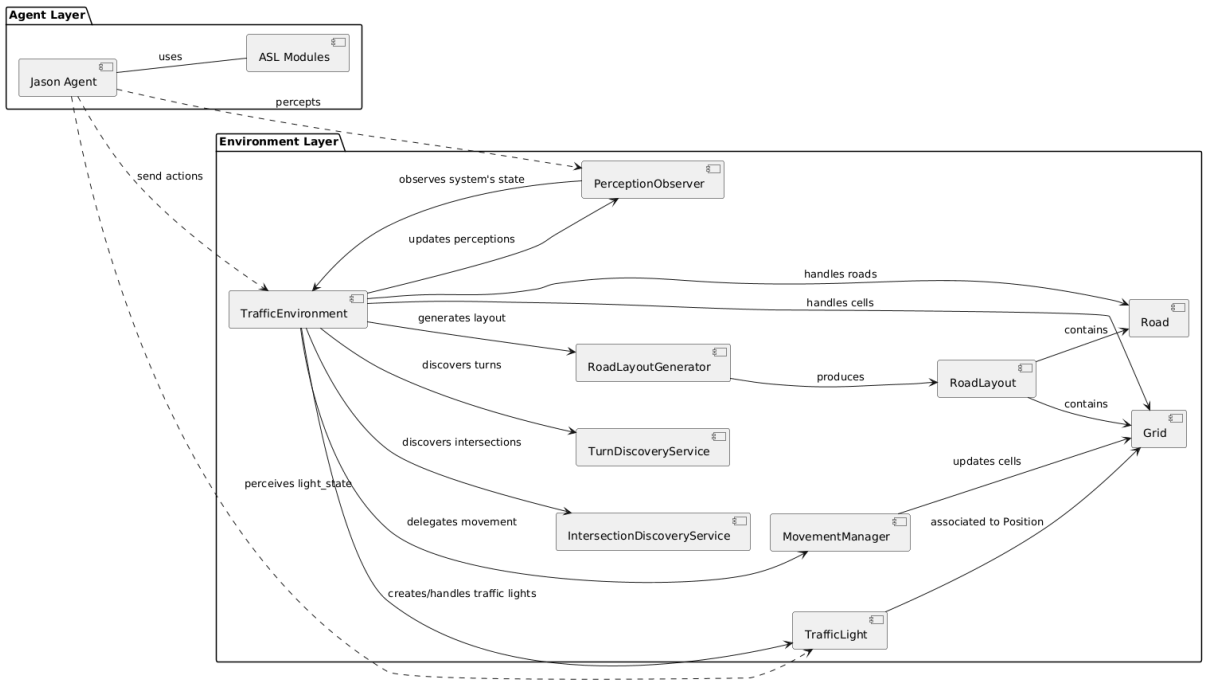
The source code architecture has been divided into two macro components, agent layer and environment layer.

Agent layer

This layer involves the jason agents, so practically, it includes the .asl modules that implement the perception, decision and coordination logic. The agents interact with the environment, and percept the system's state.

Environment layer

This layer comprehends java components that model the simulated reality. There are two main artifacts, which expose operations, observable properties and they have an internal space.
TrafficEnvironment → main artifact, works as a shared space for the multi-agent system;
TrafficLight → artifact that handles and coordinates traffic lights, each traffic light has its own timer.



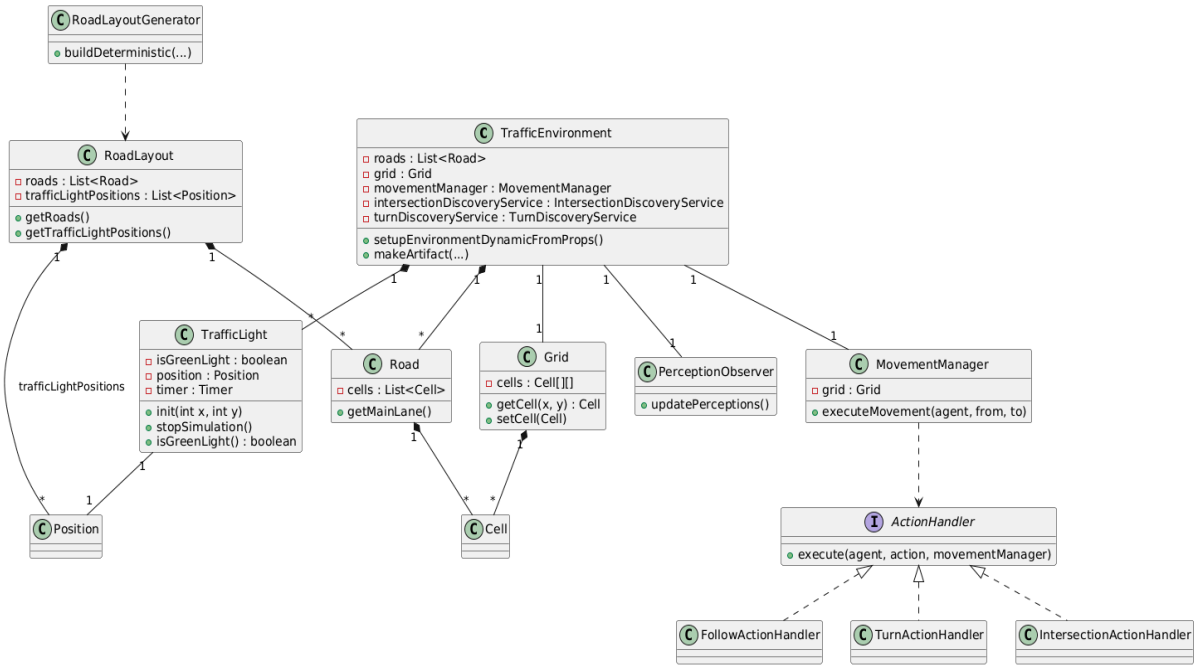
3.2 Design patterns

As can be seen in the image above, several design patterns have been used, in order to simplify and organize work better. These are the main patterns:

- Observer pattern, used to listen to perception and for turn and intersection discovery, since every agent has its own knowledge, instead of a global understanding;
- Factory pattern, used for roads generation;
- Strategy pattern, for the action handlers, in order to separate concerns.

3.3 Modelling

This section explores the modelling choices, in order to identify the main classes, events and domain entities.



Domain entities

- Vehicle agent;
  - jason BDI agent, it perceives the environment, plans and acts.
- TrafficEnvironment;
  - main artifact, handles the grid, the vehicles positions, map generation and perceptions propagation;
- TrafficLight;
  - autonomous artifact that represents a semaphore, handles its own state using a timer and notifies agents about changes.
- RoadLayout;
  - Data structure that represents the main environment components, it's used to store and give the possibility to retrieve the layout generated by the RoadLayoutGenerator class.

- Grid;
    - represents the grid map, it’s a (nxn) map, each cell is represented by the Cell class
- MovementManager;
    - handles movement actions, calculates and updates positions;
- PerceptionObserver;
    - observes environment state and updates perceptions;
- IntersectionDiscoveryService/TurnDiscoveryService;;
    - services that contains utils methods that help the jason agents to discover new intersections and turns, using listeners for support

Domain events

- Vehicle spawned → new vehicle creation;
- vehicle moved → agent changes position in the grid;
- vehicle stopped → vehicle stops because of obstacle or traffic;
- traffic light changed → from red to green or vice versa;
- intersection discovered → agent discovers a new intersection;
- turn discovered → agent discovers a new turn;
- perception updated → agent receives a new perception from the environment.

4.0 In depth analysis

This chapter focuses on the logical intuitions/implementation aspects, giving an in-depth overview, so that the MAS can be understood in its details.

4.1 Dynamic map generation

Since this project, when extended, will need a strong experimental phase, it was needed to have a map generation component, in order to generate multiple/technically infinite environment configuration. The map creation happens in a parametric and automatic way at the start, avoiding static configurations (by default, since even static parameters can be used).

4.1.1 Main components

The considered code consists of only 2 classes, RoadLayoutGenerator and RoadLayout. The first one it's responsible for the map generation, the second one it's needed to save, set and retrieve the components generated by RoadLayoutGenerator, in order to pass that information to the Artifact.

4.1.2 Generation algorithm

The algorithm has the following entry point and signature:

```
public RoadLayout build(Grid grid, SimulationEnvironmentParams params) {
    return params.getGenerationType() == SimulationEnvironmentParams.GenerationType.DETERMINISTIC
            ? buildDeterministicLayout(grid, params)
            : buildRandomizedLayout(grid, params);
}
```

In particular, considering the SimulationEnvironmentParams object, those are the needed informations:

- grid (contains also its size);
- generation type (deterministic/randomic);
- number of vertical and horizontal roads;
- spacing between roads;
- seed;

As it can be seen on the image, the method can generate a deterministic layout (number of elements generated can be chosen) or a randomized layout (even the number of elements generated is randomic).
It's important to specify that the map needs complete road connectivity, in such a way that there are no spurious roads. In order to obtain the complete connectivity, a bfs algorithm such that generation takes place only if the reached cell roads are the same number of the reachable ones.

4.1.3 Generation algorithm phases

1. **Road bases selection;**
    a. The roads starting coordinates (chosen either regularly or randomly) will be generated with the selectBases method, which will assure that all the roads starting points will be evenly spaced, based on a minimum Manhattan distance. This prevents overlaps and ensures a uniform or variable distribution of roads.
2. **Road placement;**
    a. Once the bases are defined, for each base a Road will be created with the custom Road factory method. Roads can be generated vertically or horizontally, and each one of them has two lanes, with opposite directions.
3. **Intersection placement;**
    a. Not only roads will be placed, so where a vertical and horizontal road overlap, an intersection is created, by adding 4 footprints in order to make agents cross the road in every possible direction;
4. **Generation of standalone turns;**
    a. with similar logic to intersection placement, even standalone turns will be added to the map;
5. **connectivity check.**
    a. As mentioned in the previous sub-section, it's verified that all the roads are connected to each other, to avoid unreachable cells. If necessary a central road is added to ensure connectivity.

In the algorithm implementation, for each phase, private methods are being used to maintain a good generation, good spacing and to avoid unpleasant situations, since the environment has to be perfect. In particular:

- patchCenterDirections → modifies the directions of the central cells at intersections to correctly reflect the presence of multiple traffic directions.
- generateStandaloneTurns, createHorizontalTurn, createVerticalTurn → each of these methods add map elements away from intersections, increasing the variety of the road network. They verify that the positions are valid and not too close to intersections. Unique keys are used to avoid duplicates;
- expandForbidden → expands the zones around intersections, marking cells that cannot host traffic lights or turns, according to the minimum distance parameter;
- areRoadsAllConnected, markRoadCellsAndFindStart, bfsReachableCount → these methods ensure that all roads are connected to each other:
    - markRoadCellsAndFindStart identifies all road cells and finds a starting cell.
    - bfsReachableCount performs a bfs to count how many road cells are reachable from the starting cell.

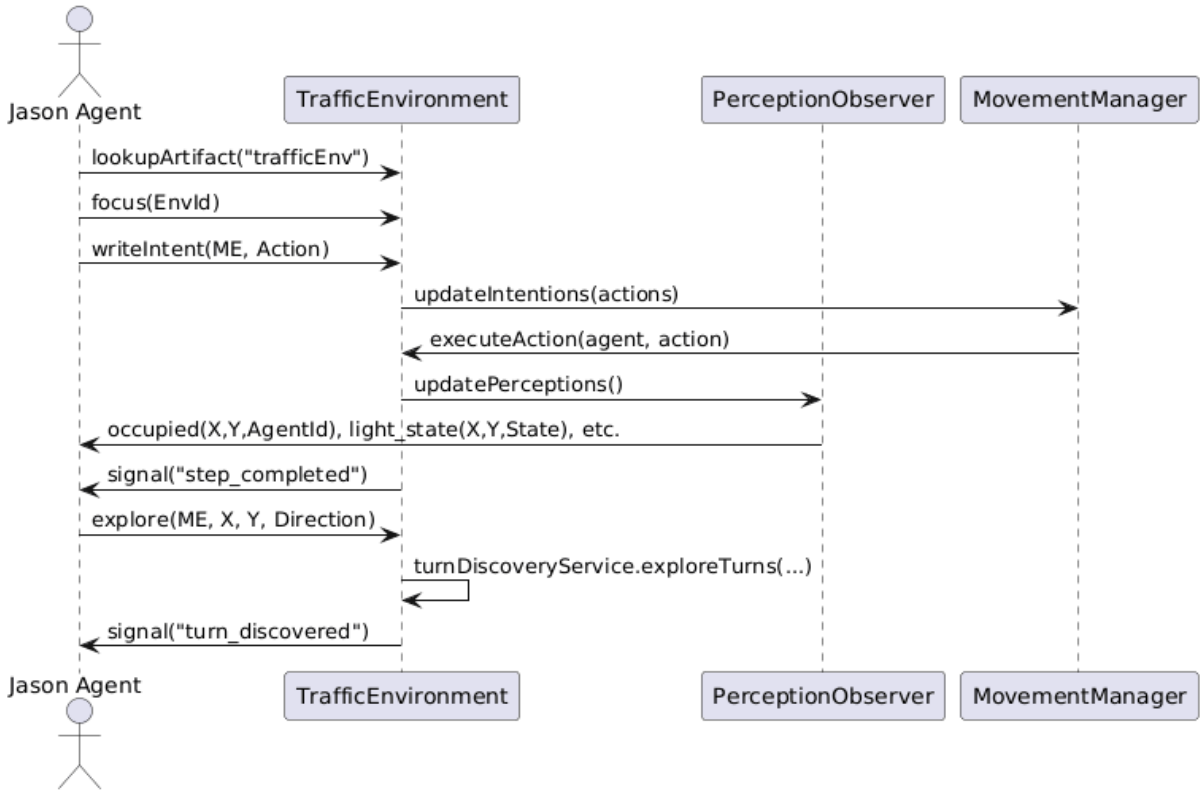
4.2 Agent behaviour and interaction

It’s really important to analyze and understand the interaction between an agent and the Artifacts, in this case, the environment (called TrafficEnvironment).
Each jason agent, has:

- beliefs → local informations (position, semaphore state, intersections, other agents position);
- goals → like cross an intersection, follow a route or coordinate;
- plans → operative rules, separated in different modules like movement, perception and coordination that handles specific behaviour aspects.

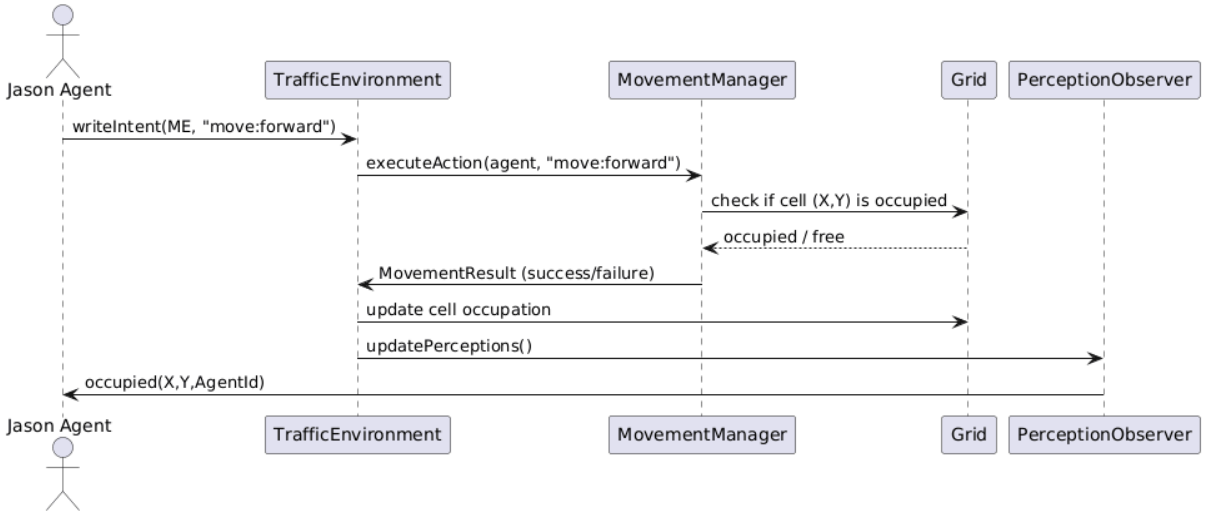
Agents perceive agent state and information via observable properties and signals made from java artifacts. In this case the perception.asl module interprets the percepts and updates the beliefs, like: light state, cell occupied or intersection discovered.
In simpler terms, agents receive feedback from the environment, update beliefs and choose their strategy accordingly. Each time an agent (vehicle) wants to move, it writes its intention that will be elaborated from the environment with the writeIntent method, defined in the artifact.
The following image, shows a possible interaction flow, in which PerceptionObserver and Movement manager are cited:

- movementmanager: centralizes and validates agent movements, ensuring safe and coordinated updates to the grid state
- perceptionObserver: aggregates and propagates environment state changes, enabling agents to perceive and react to the dynamic simulation.



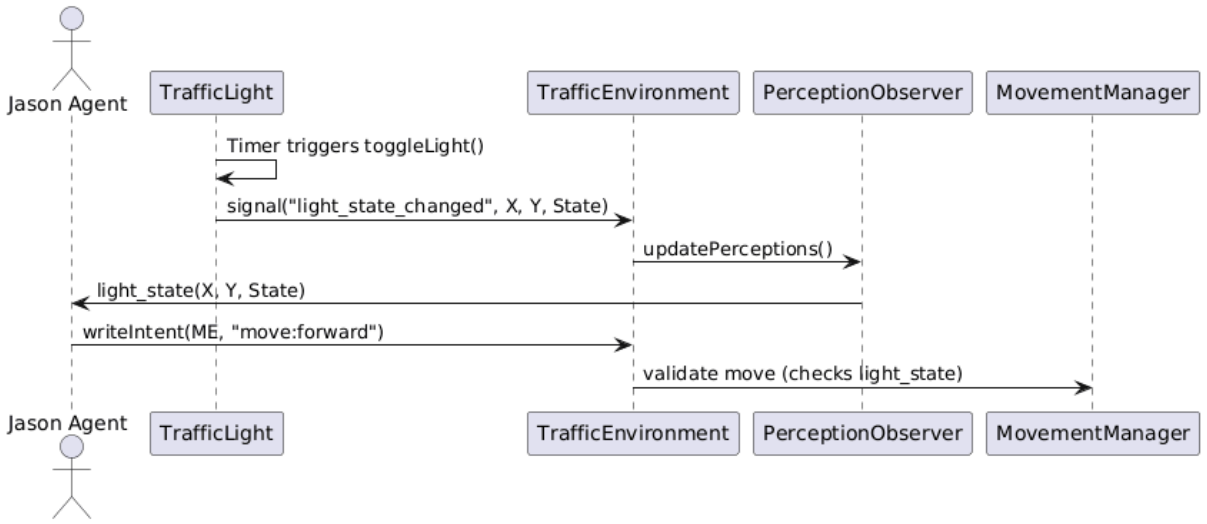
4.2.1 How agents deal with an occupied cell

When multiple agents navigate a shared environment, the management of occupied cells in the grid becomes crucial for collision avoidance and effective coordination. The interaction between movement validation and perception ensures that agents are always aware of the current occupancy of the grid, allowing them to adapt their strategies dynamically as the simulation evolves. This mechanism is possible through the occupied percept, that contains a list of occupied positions at time t of the simulation. Agents can retrieve that information to check if they're able to reach a specific position. The following sequence diagram shows how the agents deal with this specific case:



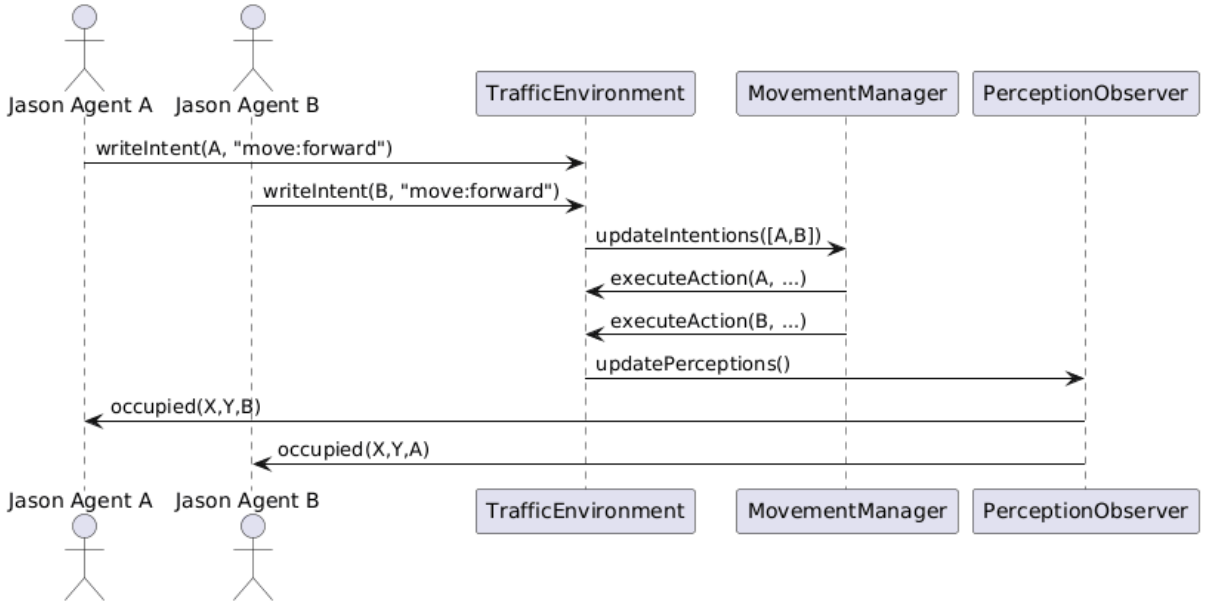
4.2.2 How traffic lights work

TrafficLight is the artifact responsible for the semaphores active in the map, each one has its own timer and can be found in the grid. This sequence diagram shows how the interaction works between agent and artifact.



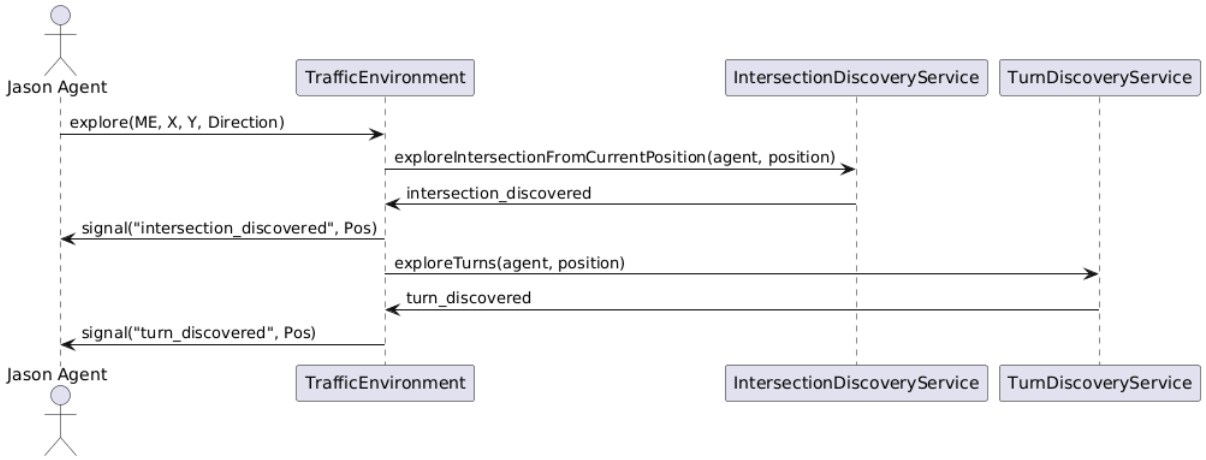
4.2.3 Agent coordination via environment

It’s fundamental to avoid collisions, intent overlap and race conditions. This is the reason why a specific workflow has been designed in the project, as it can be seen on the following sequence diagram:



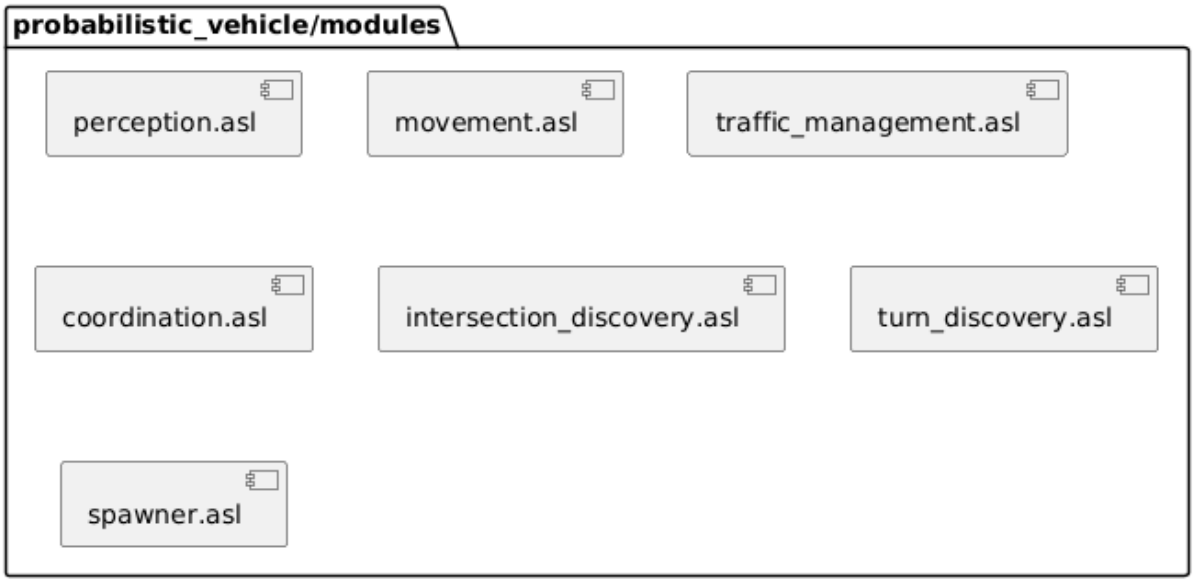
4.2.4 Turn and intersection discovery

Turns and intersections are not known apriori, it’s because, since it’s a self-organizing system, agents should be able to move based on their own knowledge. The project’s characteristics and specifications require a specific strategy in order to gain and share knowledge. Agents, as can be seen in the following sequence diagram, explore in order to find intersections, or turns, once they find them (when they are close to them), they share their knowledge to the shared space, defining a belief.



4.3 Jason module architecture for autonomous agents

This section will focus on explaining the jason code architecture, in order to understand how the agents are built and how they react to external stimuli. In order to conserve separation of concerns and make it more scalable, code has been organized in several, different modules, based on their utility, that will be explained in detail in the current section.



4.3.1 Movement module

The movement module encodes the agent's reasoning for navigation and movement, inside the defined environment grid. It's a simple yet important module, and these are the main responsibilities:

- compute and calculate next position → grid's dimensions are used to implement toroidal movement, to avoid agents getting stuck;
- occupancy check.

Before moving, the agent checks, by reading the occupied perception, if the cell is free or occupied.

```
+!execute_move(NextX, NextY) : get_name(ME) <-
    ?occupants(List);
    .print("[DEBUG][Jason] occupants: ", List);
    if (not .member([NextX, NextY], List)) {
        writeIntent(ME, "follow");
    } else {
        .print("[wait] Cell (", NextX, ",", NextY, ") occupied, waiting");
        writeIntent(ME, "wait");
    }.      You, 4 weeks ago • refactor: update asl code structure
```

4.3.2 Perception module

The perception module contains the code that processes percepts received from the environment artifact, in order to maintain the agent's state up to date. Its main features are:

- Positional tracking;
- occupancy awareness;
- traffic light change detection;
- simulation percept.

Code:

```
You, 5 days ago | 1 author (You)
+at(ME, X, Y)[source(percept)] : get_name(ME) <-
    -at(ME, _, _);
    +at(ME, X, Y).

+occupied(X, Y)[source(percept)] <-
    -occupied(X, Y);
    +occupied(X, Y).

+occupants(List)[source(percept)] <-
    -occupants(_);
    +occupants(List).

+direction(X, Y, Direction)[source(percept)] <-
    -direction(X, Y, _);
    +direction(X, Y, Direction).

+light_state(X, Y, State)[source(percept)] <-
    -light_current_state(X, Y, _);
    +light_current_state(X, Y, State);
    .print("[traffic_light] has seen traffic light at (", X, ",", Y, ") = ", State).

+light_state_changed(X, Y, State)[source(percept)] <-
    -light_current_state(X, Y, _);
    +light_current_state(X, Y, State);
    .print("[traffic_light] noticed that traffic light at (", X, ",", Y, ") changed to ", State).

+interval(I)[source(percept)] <-
    +simulation_interval(I).
```

4.3.3 coordination module

As the title suggests, the coordination module contains methods that work toward the negotiation of movements, in order to avoid conflicts when moving in a shared space. It has only one, but important method, that checks conflicts.

```
+!check_coordination(NextX, NextY) : get_name(ME) <-
    if (agent_intention(OtherAgent, NextX, NextY, _) & OtherAgent \== ME) {
        .print("[wait] ", OtherAgent, " at (", NextX, ",", NextY, ")");
        writeIntent(ME, "wait");
    } else {
        !check_traffic_light(NextX, NextY);
    }.
```

4.3.4 intersection discovery module

The intersection discovery focuses on the exploration of intersections (same thing happens with turns), in order to gain knowledge to share with other agents.
When an agent perceives a new intersection it adds the location to its beliefs, and as introduced before, it shares the knowledge via intersection_discovered belief.

```
You, 3 weeks ago | 1 author (You)
+intersection_available(X, Y)[source(percept)] : not known_intersection(X, Y) <-
    +known_intersection(X, Y);
    .print("[intersection] discovered (", X, ", ", Y, ") ").

+intersection_discovered(Agent, X, Y)[source(percept)] :
    get_name(ME) & Agent \== ME <-
    +known_intersection(X, Y);
    .print("[intersection] learned from ", Agent, ": (", X, ", ", Y, ")").

has_available_intersections(X, Y) :- known_intersection(X, Y).

+?get_available_intersections(X, Y, L) <-
    if (known_intersection(X, Y)) { L = [[X, Y]]; } else { L = []; }.
```

4.3.5 spawner module

Spawner module helps the random generation, according to the roadLayoutGenerator, presented in the second chapter of the report.

```
!start <-
    lookupArtifact("trafficEnv", EnvId);
    focus(EnvId);
    getSimAgents(NofAgents);
    getSimSeed(Seed);       You, last week • feat: create spawner class
    .print("[spawner] spawned.. agents: ", NofAgents, " seed: ", Seed);
    !spawn_loop(1, NofAgents, Seed).

!spawn_loop(I, N, Seed) <-
    if (I <= N) {
        pickRandomFreeRoadCell(Seed, X, Y);
        if (X \== -1 & Y \== -1) {
            .concat("vehicle", I, Name);
            Params = [name(Name)];
            .create_agent(Name, "probabilistic_vehicle/complete_vehicle.asl", Params);
            .send(Name, tell, name(Name));
            placeAgent(Name, X, Y, Result);
            .send(Name, tell, env_ready);
            .print("[spawner] ", Name, " position: ", X, ",", Y, " result=", Result);
            .wait(1000);
            I1 = I + 1;
            !spawn_loop(I1, N, Seed)
        } else {
            .print("[spawner] no free cell, retry...");
            .wait(300);
            !spawn_loop(I, N, Seed)
        }
    } else {
        .print("[spawner] done.");
    }.


!spawn_loop(I, N, _) : I > N.

!start.
```

5.0 Conclusion & possible improvements

The project involves the creation of a modular and extensible multi-agent system, this demonstrates how distributed agents can achieve safe and efficient navigation through local perception, and the possible efficiency of cognitive stigmergy approach. This can offer a solid fundation for future works, in order to understand better how this can be useful in real-life (and possible real time) applications.
A lot of work can still be done, such as:

- adding a graphical interface;
- metrics;
- experiments and statistical significance between classical and cognitive stigmergy controllers;
- adding more features or agents;
    - pedestrians (that can also cross roads);
    - signals;
    - possible crashes;
    - speed.
- adding environmental factors, like weather, maybe based on real-life conditions;
- scalability tests;
- advanced algorithms;
- path calculation algorithms, like a* algorithm.