City Dynamics: Multi-Agent Simulation

Alice Turrini
alice.turrini@studio.unibo.it

Maud Ravanas-Deshays
maud.ravanasdeshays@studio.unibo.it

July 2024

This project aims to simulate the traffic flows in a city by means of a multi-agent system. The main characters of our city are cars, that drive in the streets, and pedestrians that walk within buildings. They rely on their perception of the environment to achieve specific goals. These agents will be autonomous and intelligent, capable of decision making and interaction.
The implementation of this system is based on the JaCaMo development platform[6], where Jason is used as the reference agent programming language to implement agents driven by their beliefs and their goals (BDI architecture).

1 Goals

The project is focused on creating seamless traffic flows involving different types of agents with specific behaviors inside a city environment.

1.1 Environment requirements

In any multi-agent system, agents are situated in an environment that can be dynamic and unpredictable.
In our case, the environment is a city consisting of different elements:

- **Buildings:** These will represent the main places where pedestrians can freely walk because cars are not allowed inside. There are four specific buildings: a supermarket, a school, a park and an office, which are targets of some agents. Some buildings are disconnected from each other and the only possible connection for pedestrians is through zebra crossings to safely cross the roads.
- **Streets:** These are mainly the location of car agents; indeed pedestrians are not allowed to walk on them with the only exception of the zebra crossing blocks.

Each street lane has an assigned direction (or two in case of crossroads) and some of them have also precedence. Cars will have to take all these information into account when moving to follow the traffic regulations.

- **Zebra Crossings:** These are some specific street blocks, that allows pedestrians to cross lanes to reach a different building. To avoid regrettable accidents, cars always check that there is not a pedestrian on these blocks before driving on them.
- **Garage:** This is the parking spot of the helicopter, an agent with unique characteristics that repairs broken cars.

1.1.1 Building visualisation

Here are shown the different building blocks of the infrastructure and their visualisation:

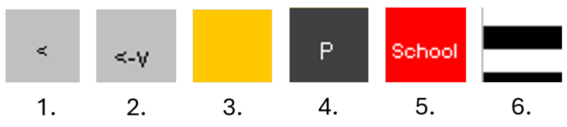


Figure 1: 1. unidirectional street;
2. bidirectional street (crossroads);
3. building;
4. parking of the helicopter;
5. special building (school, supermarket, office or park);
6. zebra-crossings

In addition to these specific structures, the environment also creates some unpredictability in the system by making the cars randomly breaking down from time to time.

1.2 Agents requirements

Now that the stakes of the context in which our agents should operate are defined, let's talk more in depth about our agents. In a multi-agent system, agents are immersed in a society of other agents with who they have to coexist and interact.
We defined four different types of agents:

- **Cars:** They drive autonomously around the city, if they are in street lanes they need to follow the associated direction, and instead in crossroads they can choose where to go, thanks to bidirectional streets, following the constraint of precedence between different lanes.
  An other very important constraint is that two cars cannot share the same position as it would be similar to an accident.
  If they come near an obstacle, that could be a dead-end, another car, or a building block, they will randomly choose another direction to take, with the only constraint

of making sure to not go back.
In zebra crossing, as assumed in real life, if a pedestrian is in front of the car, the latter waits for the pedestrian to have cleared the way before moving on.
As said before, sometimes a car breaks down, and in that case, it will stop moving until it calls for help and gets fixed by the repair helicopter.

- **Pedestrians:** They walk inside buildings and between them using zebra crossings. Contrary to cars, multiple pedestrian agents can share the same location. Their goal is to live their life doing some specific tasks going towards specific buildings, that change with the agent's type:
    - adult pedestrians will first go to the office, and then to do the grocery shop at the supermarket;
    - child pedestrians instead will first go to the school, and then to play at the park.

  Once they complete their path, they will stop moving because they finish their day.

- **Helicopter:** This agent is a special vehicle that doesn't have to follow any kind of traffic regulations nor avoid buildings because it can fly: this will enable to avoid congestion and arrive faster to the broken car.
  It will start parked at the garage and will go to a broken down car as soon as it gets its SOS message. Once the helicopter arrives at the rescue, it will repair and then go back to its parking place. If another car calls for its help while it is going back to its garage, it interrupts its way back to directly go repair the other car.

1.2.1 Agents visualisation

Here are shown the different agents of the infrastructure and their visualisation:

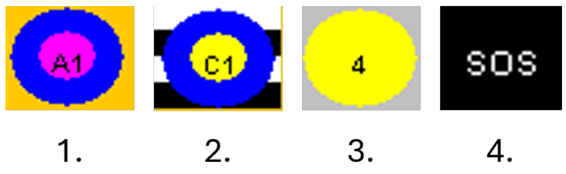


Figure 2: 1. adult pedestrian; 2. child pedestrian; 3. car;4. repair helicopter

1.3 Coordination

While each agent has its own individual goals, the requirement to have a system simulating seamless traffic flows implies that agents are able to interact with each other so as to coordinate their actions and achieve together this common goal of harmony.
In our case, the city environment works as a mediator, providing the agents with useful percepts so that they can regulate their behaviours. Communication, as we will see in

the following part, also enables a better coordination of our agents.
Two scenarios of coordination are considered:

- **Cars movements:** A car has a rather constrained way of moving as it will not be able to go in buildings, must follow traffic regulations, and most of all must not collide with other cars and or pedestrians. Looking at its percepts to get information about its surrounding, a car agent will hence have to be able to stop its movement and wait if a pedestrian is crossing, or to change of direction so as to not create congestion if there is another car blocking the way.
- **Crossroads:** are particular situations of bidirectional blocks in streets, where cars have to choose the direction and potentially cross another lane. As already mentioned some streets have precedence, and so if a car is in one of them, it will announce which location it is targeting telling that it has the priority. The other cars will have to take this information to let that vehicle pass first in case of conflicts.

1.4 Communication

In a multi-agent system, communication is a key factor as it is the mean agents use to share information and data between themselves. Indeed, agents are autonomous and encapsulate their own flow of control, which means that under no circumstances can an agent directly control another agent.
Two scenarios of communication are considered:

- **Helicopter fixing Car**: The first one will be the trigger to the repair process, as a car will have to ask the repair helicopter for help and send its position to it so that this latter can come fix it.
- **Pedestrian greetings**: The second scenario will be that pedestrians, when they meet each other on their way to their target buildings, will greet each other. Adults will have the role of a ping, and children of pong. This means that the communication will only occur between an adult and a child, not between two adults or two children.

2 Design

2.1 BDI Architecture

The system we want to implement is goal-directed since each agent has its own purpose. So as to reach this latter, agents must behave both proactively (deliberating according to their mental representation of the world) and reactively (adapting their behaviour to the environment's changes).
We based our work on the BDI architecture[4], which stands for Beliefs, Desires and Intentions. In this architecture, agents try to reach their goals (desires) by performing specific tasks (plans) according to what happens around them and what they believe in, they have knowledge about themselves and the environment surrounding them.
This architecture enables agents to actually reason and decide what workflow to follow to achieve a certain goal, while taking into account the changes of the context they are in (dynamic environment, other agents, etc.). Because of the way this definition of autonomy actually integrates intelligence in some extent, one could say we implemented strong agents.

2.2 Technology used: JaCaMo

To implement and manage our system, we decided to use the JaCaMo development platform[6]. In this framework, Jason[7] is used as the reference agent programming language. This Java-based language extends the AgentSpeak[3] abstract language and hence allows to concretely program BDI agents.
For the environment, there is Cartago[5], a framework that provides an infrastructure for creating, managing, and interacting with artifacts in a shared custom environment. These artifacts are tools that agents can manipulate through specific Java-based operations in order to perform pragmatical actions on our custom environment.
Finally, the "Mo" in JaCaMo stands for Moise[8], an organizational model which enables to use notions of groups, missions and roles in MAS. We actually did not use this model in this project as it was not necessary to answer the needs we had defined. However, for a project like the Gold Miners[1] one, that we referred to for understanding how to manage the grid visualisation with JaCaMo, one can note that the authors had used Moise so as to define different teams of miners.

2.3 Design pattern: MVC

The project has been constructed and designed to be modular as the main design choice. We wanted each functionality to be added after the others without overturning everything. This is one reason for which we decided to follow the Model-View-Controller framework in the project.

To give our project some coherence indeed, we choose to use the MVC design pattern, because it provides a clear separation of tasks and responsabilities, making the code and the whole application easier to manage. As expected it was also easy to update with new functionality one after the other.

Here there is the representation of our structure with the corresponding files and the most important methods.

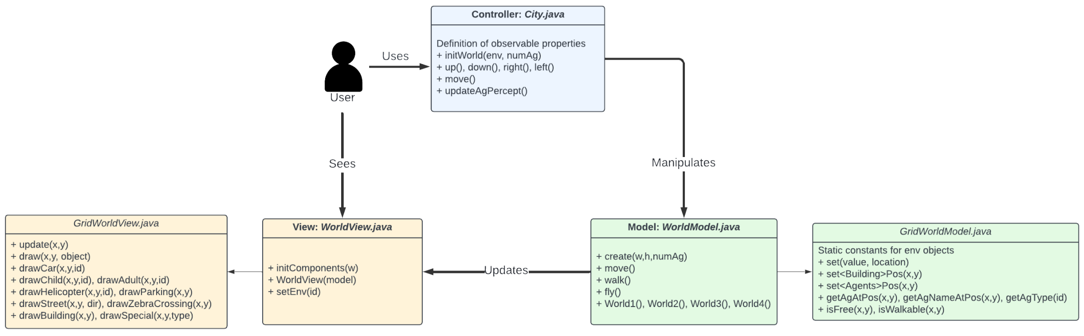

Controller: City.java
Definition of observable properties
+ initWorld(env, numAg)
+ up(), down(), right(), left()
+ move()
+ updateAgPercept()
User
Uses
Sees
Manipulates
GridWorldView.java
+ update(x,y)
+ draw(x,y, object)
+ drawCar(x,y,id)
+ drawChild(x,y,id), drawAdult(x,y,id)
+ drawHelicopter(x,y,id), drawParking(x,y)
+ drawStreet(x,y, dir), drawZebraCrossing(x,y)
+ drawBuilding(x,y), drawSpecial(x,y,type)
View: WorldView.java
+ initComponents(w)
+ WorldView(model)
+ setEnv(id)
Updates
Model: WorldModel.java
+ create(w,h,numAg)
+ move()
+ walk()
+ fly()
+ World1(), World2(), World3(), World4()
GridWorldModel.java
Static constants for env objects
+ set(value, location)
+ set<Building>Pos(x,y)
+ set<Agents>Pos(x,y)
+ getAgAtPos(x,y), getAgNameAtPos(x,y), getAgType(id)
+ isFree(x,y), isWalkable(x,y)

Figure 3: Diagram showing the MVC structure of the files and most important methods

2.4 The Different Worlds

The project contains four diworlds: each of them is linked to a specific map and a number of different types of agents. This choice was made to have the possibility to see particularly features in several diverse contexts.

- **World 1:** There are 4 big blocks of building separated by a street with 2 lanes that creates a crossroads in the middle. Zebra crossing, the 4 different types of buildings and the helicopter are present in the environment. The agents playing a role here are 2 cars, one child, one adult and the helicopter.

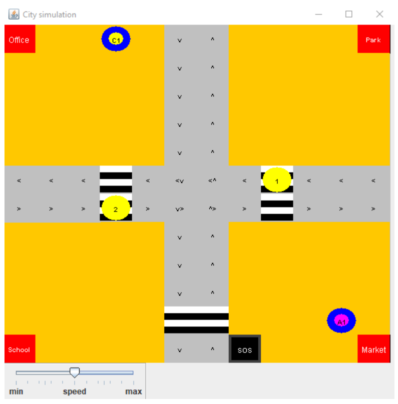


Figure 4: World1

- **World 2:** The simplest world to better test the interaction between cars and pedestrians on the zebra crossings. Indeed it is just a horizontal street with 2 lanes that divides the building blocks in an upper and lower side. In this case, there is no helicopter, but there are 4 cars and 4 pedestrians: this increases the possibilities of interactions.

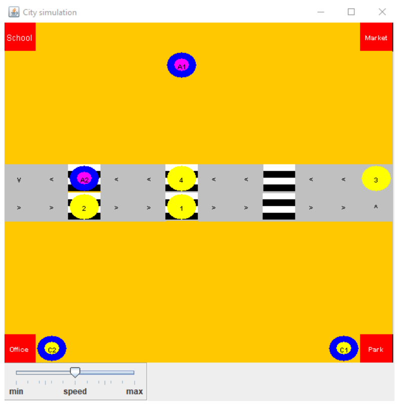


Figure 5: World2

- **World 3:** It's very similar to the World 1 but it also have a street with 2 lanes all around the city that surrounds all the buildings. Zebra crossing, the 4 different types of buildings, and the helicopter are present in the environment. The agents involved here are 4 cars, 2 pedestrians, and the helicopter.

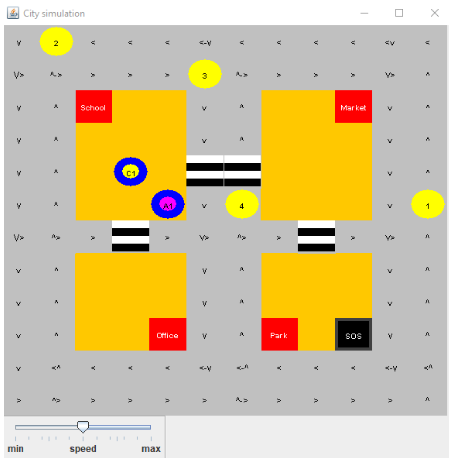


Figure 6: World3

- **World 4:** Same as World 3 but with two more pedestrian to better analyse the communication between them.

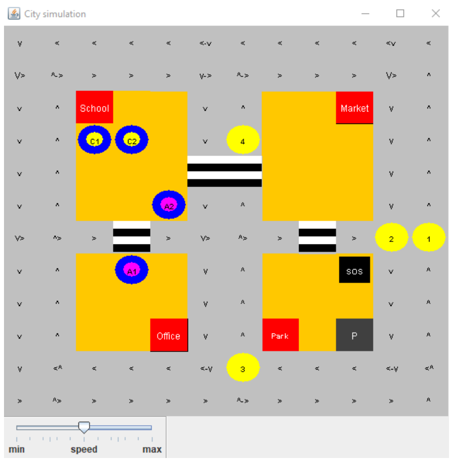


Figure 7: World4

2.5 Management of defined objects

Inside the worlds, that are grids, there are multiple objects having different roles, and some of them can also share the same cells of the grid. We decided to represent each object with a bitmask, so in this way it is possible to use binary operator as & and | to differentiate multiple objects inside a block.
As it is shown in the GridWorldModel.java all the objects are constants power of 2, for creating easily the bit mask, here it is an example of a pedestrian child crossing a street on the zebra crossing:

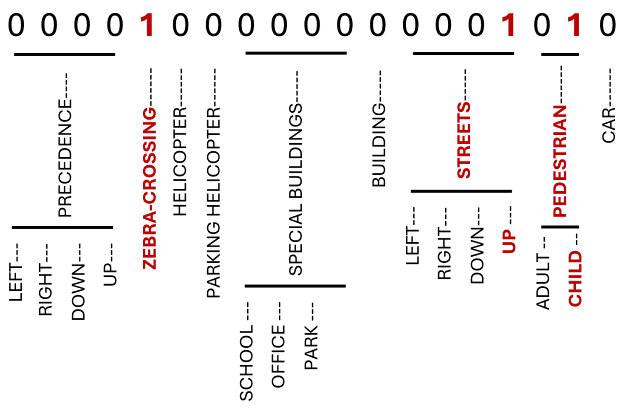


Figure 8: Bit mask of a child crossing a street up on a zebra-crossing

Indeed, all these operations would return a boolean "true":

$$\begin{aligned} & (\text{cell}_{x,y}\,\&\,\text{ZEBRA\_CROSSING}) \neq 0 \\ \text{and} \quad & (\text{cell}_{x,y}\,\&\,\text{PEDESTRIAN\_CHILD}) \neq 0 \\ \text{and} \quad & (\text{cell}_{x,y}\,\&\,\text{STREET\_UP}) \neq 0 \end{aligned}$$

3 Implementation: Agents Behaviors

Based on the requirements defined earlier and the design choices made, we implemented the different agents of our system. Here is an explanation of the programming and logical flow each type of agent follows so as to perform the different behaviors enabling it to reach its goals.

3.1 Cars

As a way to simplify their behaviour, we first of all decided to represent cars as simple circles, making it unnecessary to introduce any notion of relative orientation there.

- **Initial goal: !drive_random**
  As the name states, cars main purpose is to randomly drive while following traffic regulations. To do so, they first observe two percepts that the environment makes available to them: their current position (*pos*), and the direction assigned to the cell they are currently on (*cellC*). Given the context, the cars then try to move in one of four directions: up, down, right and left.

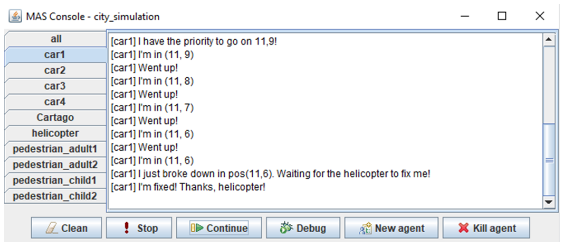
  These actions internally call for the method *move* that checks the bitmask of the cell targeted by the car, and there are two possible scenarios:

  - The cell is eligible for the car to move to it: it means it is a street and no other agent occupies it. Then the car is deleted from the current cell and added to the destination cell. The information that the move was a success is then returned to the car agent through a percept *success*, and it then continues randomly driving cell after cell.
  - The cell is not eligible: another car occupies it, or it is a building. Then the move function returns a boolean indicating the failure through the percept *fail*. This triggers the *!change_direction* plan which calls for a custom internal action that draws a new random direction. The car agent can then try to go in new direction.

  Note that if the car is on a bidirectional block, so it's in a crossroads, it randomly chooses between the two directions thanks to the custom internal action *random_direction* (the same one used to change direction).

- **Breaking down:**
  In a multi-agent systems, agents have to deal with the unpredictability of the dynamic environment. In our case, this comes from the random breaking down of

cars. To implement this, the move function in City.java has been enriched with a random section, that makes the movement of a car fail with a chance of 1 out of 50. If the car breaks down, the environment updates its percept *state*, from *'works'*, to *'broken_down'*. Since the movements the car are conditioned by the *state == 'works'*, a broken down car won't move anymore.
Moreover, the update of state percept is the trigger to the asynchronous communication act: the car sends an SOS message to the repair helicopter through an "achieve" illocutionary plan. During the communication for help, the current coordinates of the car are also shared so that the helicopter knows where to go to fix it. Once the helicopter is done fixing the car, the percept will be updated as *state == 'works'* again and the car is then able to resume moving normally.



```
[car1] I have the priority to go on 11,9!
[car1] I'm in (11, 9)
[car1] Went up!
[car1] I'm in (11, 8)
[car1] Went up!
[car1] I'm in (11, 7)
[car1] Went up!
[car1] I'm in (11, 6)
[car1] Went up!
[car1] I'm in (11, 6)
[car1] I just broke down in pos(11,6). Waiting for the helicopter to fix me!
[car1] I'm fixed! Thanks, helicopter!
```

Figure 9: Logs of a car breaking down

- **Zebra crossings:**
  At zebra-crossings, if a pedestrian is in front of the car, this latter will perceive it through the observable percept who⟨initial of the position of the cell⟩ from the environment.
  Given that the pedestrian is an agent, so an obstacle for the car, it could trigger the *change_direction* plan. However that is not the adequate behaviour: the car has to wait at the zebra-crossing for the pedestrian to free the way and then keep going, as in real life.
  This is done thanks to the percept fail: the first argument indicates which movement failed (for example, "left"), and the second argument is a boolean to know if the reason for the failure was because of a pedestrian or not. So if the car gets *fail(left, True)*, it means that it faces a busy zebra-crossing block. In this case it will simply wait for a while and then try to move again. If the pedestrian is still there, this movement will fail too, and the car will try and try again until the road is free.

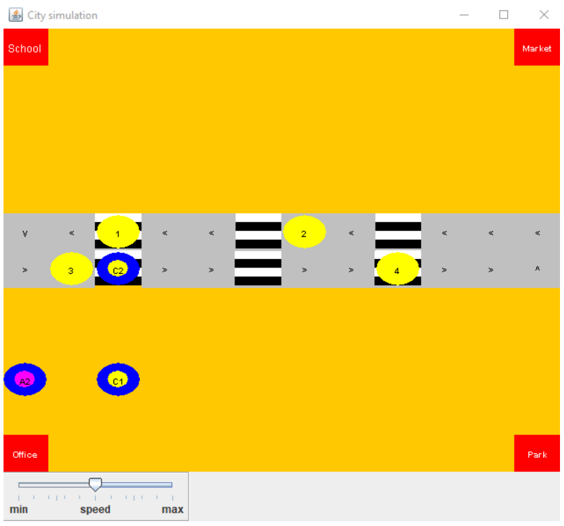
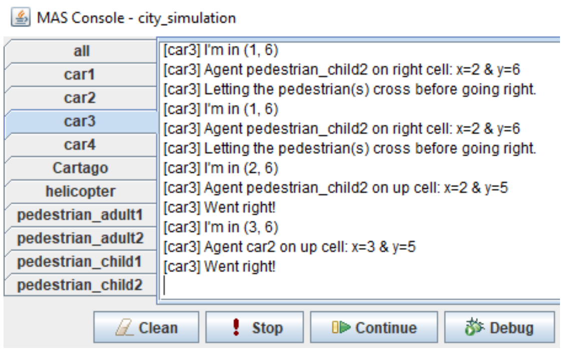


Figure 10: Visualisation of car3 letting child2 pass



```
[car3] I'm in (1, 6)
[car3] Agent pedestrian_child2 on right cell: x=2 & y=6
[car3] Letting the pedestrian(s) cross before going right.
[car3] I'm in (1, 6)
[car3] Agent pedestrian_child2 on right cell: x=2 & y=6
[car3] Letting the pedestrian(s) cross before going right.
[car3] I'm in (2, 6)
[car3] Agent pedestrian_child2 on up cell: x=2 & y=5
[car3] Went right!
[car3] I'm in (3, 6)
[car3] Agent car2 on up cell: x=3 & y=5
[car3] Went right!
```

Figure 11: Logs of car3 letting child2 pass as shown on Figure 10

- **Precedence:**
  At crossroads, represented by bidirectional blocks, cars can randomly choose between the two directions. However, in some cases, one direction among these two is indicated as the main one. The environment, with the information of the precedence assigned, builds the property *cellX* with X as the initial letter of the description of the cell. If no precedence has been assigned to the block or if it is a unidirectional one, then the term *no_precedence* is used. A car can check the main direction of the block it is currently on through the percept *cellC*. If it has the priority, it then has to share this information with the other agents.
  The problem is that the cars that may be impacted by this situation (having to give way to the car with precedence) are out of reach: an agent can indeed only get the names of the agents in the four blocks directly surrounding it. So, a targeted communication is impossible. Instead, the car with precedence rather broadcasts

the information that it has the priority, along with the coordinates of the cell it is targeting. The percept *priority(X, Y)* is hence added in the belief base of all of the agents, and is considered when trying to move. If a car is targeting a cell and that it's in its belief base in that percept, it then waits for it to pass first and then try again to move.
Note that as soon as it performed its move, the car with precedence erases the belief from the other agents belief base through an illocutionary force "untell": this enables the belief base of the agents to stay clean and relevant with the current context.

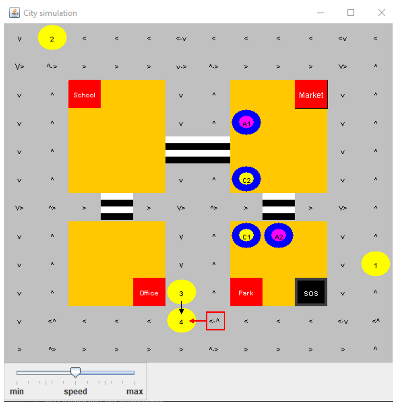
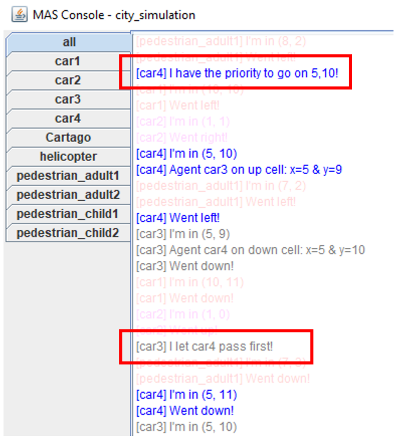


Figure 12: Visualisation of a precedence situation: car4 comes from a bidirectional block (cf. red rectangle) and chose the main direction (here, going left). It then has the priority over car3, which was also going to the same cell in (5,10) but lets the car4 pass first



```
[pedestrian_adult1] I'm in (8, 2)
[pedestrian_adult1] Went left!
[car4] I have the priority to go on 5,10!
[car1] I'm in (10, 10)
[car1] Went left!
[car2] I'm in (1, 1)
[car2] Went right!
[car4] I'm in (5, 10)
[car4] Agent car3 on up cell: x=5 & y=9
[pedestrian_adult1] I'm in (7, 2)
[pedestrian_adult1] Went left!
[car4] Went left!
[car3] I'm in (5, 9)
[car3] Agent car4 on down cell: x=5 & y=10
[car3] Went down!
[car1] I'm in (10, 11)
[car1] Went down!
[car2] I'm in (1, 0)
[car2] Went up!
[car3] I let car4 pass first!
[pedestrian_adult1] I'm in (7, 3)
[pedestrian_adult1] Went down!
[car4] I'm in (5, 11)
[car4] Went down!
[car3] I'm in (5, 10)
```

Figure 13: Logs of the precedence situation displayed in Figure 12:
the timing of percepts printing can vary depending on the system processing, hence car3 logs getting printed after car4 ones here, but it is still possible to see that car3 gave way to car4 before moving to (5,10)

3.2 Pedestrians

Let's see some intelligent behaviors of these agents.

- **Walk towards a specific place: using the AStar Algorithm**
  Whether it is a child or an adult, pedestrians' main purpose is to walk towards specific positions.
  An adult must first head towards the office, as described in the requirements, and then to the supermarket. For a child, it is rather first the school, then the park. They have access to the position of these specific buildings thanks to percepts made observable by the environment.
  When these agents are walking, the move action is called and it triggers the walk function of WorldModel which returns a success or fail boolean.
  These agents moves in an intelligent way, to know the direction to take to go to the target position they use a pathfinding algorithm, the A Star. Note that pedestrians can only walk on cells coded as either buildings or zebra crossings, so the algorithm has to find a path in the grid environments avoiding obstacles.

3.2.1 A Star algorithm

  The A Star is a searching algorithm [2], one of the most famous pathfinding algorithms. It efficiently finds the shortest path from a starting location to a target location in a given world model. It's one of the best algorithms because it combines features of uniform-cost search and pure heuristic search to achieve optimal performance.
  It creates a graph with all possible paths and it evaluates each node of this graph with the function $f(n)$, that is computed as:

$$f(n) = g(n) + h(n) \tag{1}$$

  where:

  - $g(n)$ is the cost from the start node to node n,
  - $h(n)$ is the estimated cost from node n to the goal, the heuristic.

  The heuristic computation is the core of this algorithm, it is the real intelligence of the pedestrian agents, and it has to take into consideration the obstacles during the path.

- **Synchronous communication between pedestrians**
  Communication between pedestrians is implemented when they perceive each other in adjacent cells. As stated in the requirements, adult pedestrians have the role of ping, meaning that they start the communication, and children have the role of pong, so they have to reply.
  Since we want the pedestrian to communicate in a synchronous way, we want them to stop walking when they greet someone, and to wait for the communication act

to be done before continuing their day.
We built the percept *who* so that its third argument states the type of agent encountered: car, childPedestrian, adultPedestrian or nobody. This information allows to trigger the communication plans only when adults meet children, as we only want communication between a ping and a pong to avoid interferences.
When an adult perceives a child it did not greet before (cf. the *last_greeted* percept), the event *!greet* is raised. The adult sends a message to the child to greet them, with an illocutionary force "tell", then its belief *waiting* is updated to 1, preventing it to keep moving.
On the child's side, when it receives greetings from an adult, it will stand still because its percept waiting will become 1 and again, the prerequisite for the movement plans is waiting at 0. It then sends to the adult a *greetings_back* and waits for some time before moving again. After receiving this message, the adults update its waiting belief back to 0 and then it will move again.

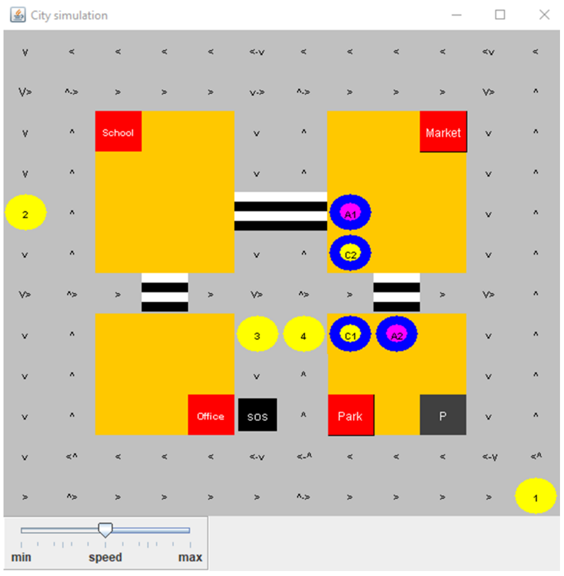
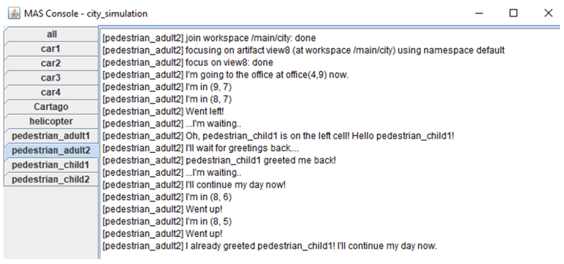
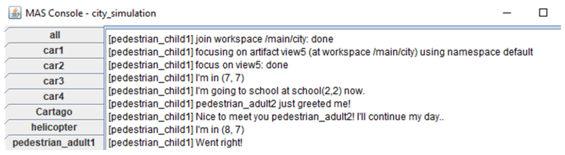


Figure 14: Communication acts between adult2-child1, and adult1-child2



```
[pedestrian_adult2] join workspace /main/city: done
[pedestrian_adult2] focusing on artifact view8 (at workspace /main/city) using namespace default
[pedestrian_adult2] focus on view8: done
[pedestrian_adult2] I'm going to the office at office(4,9) now.
[pedestrian_adult2] I'm in (9, 7)
[pedestrian_adult2] I'm in (8, 7)
[pedestrian_adult2] Went left!
[pedestrian_adult2] ...I'm waiting..
[pedestrian_adult2] Oh, pedestrian_child1 is on the left cell! Hello pedestrian_child1!
[pedestrian_adult2] I'll wait for greetings back....
[pedestrian_adult2] pedestrian_child1 greeted me back!
[pedestrian_adult2] ...I'm waiting..
[pedestrian_adult2] I'll continue my day now!
[pedestrian_adult2] I'm in (8, 6)
[pedestrian_adult2] Went up!
[pedestrian_adult2] I'm in (8, 5)
[pedestrian_adult2] Went up!
[pedestrian_adult2] I already greeted pedestrian_child1! I'll continue my day now.
```

Figure 15: Logs of the adult2 pedestrian communicating with child1 in Figure 14



```
[pedestrian_child1] join workspace /main/city: done
[pedestrian_child1] focusing on artifact view5 (at workspace /main/city) using namespace default
[pedestrian_child1] focus on view5: done
[pedestrian_child1] I'm in (7, 7)
[pedestrian_child1] I'm going to school at school(2,2) now.
[pedestrian_child1] pedestrian_adult2 just greeted me!
[pedestrian_child1] Nice to meet you pedestrian_adult2! I'll continue my day..
[pedestrian_child1] I'm in (8, 7)
[pedestrian_child1] Went right!
```

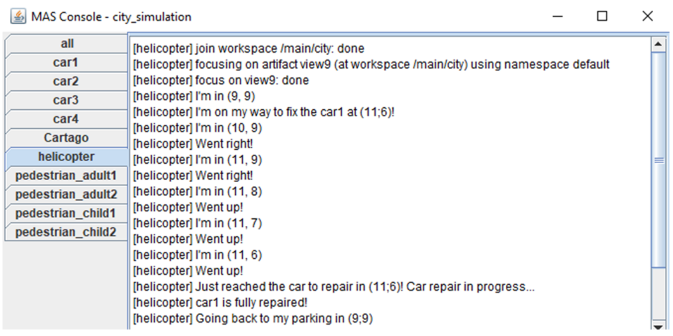
Figure 16: Logs of the child1 pedestrian communicating with adult2 in Figure 14

3.3 Helicopter

The helicopter is the repair agent in charge of fixing the broken down cars. As a general information, if no helicopter is instantiated, there won't be any random breaking down of the cars. Important to specify that only one helicopter must be instantiated, so that cars can use its name in communication acts with no risk of getting it wrong.

- **Movement of the helicopter:**
  The helicopter can fly, the specific fly action is less constrained than the others movement functions: indeed, the helicopter agent can go on top of other agents, buildings and streets.

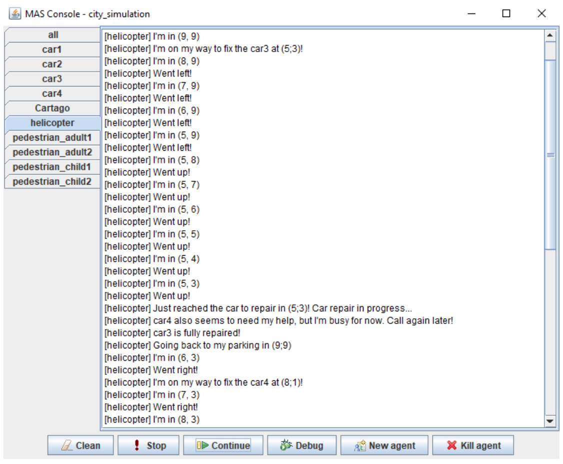
- **Goal upon message reception: !fix_car(X,Y)**
  As explained above, when a car agent breaks down, it sends the helicopter a message to call it for help and indicate to the repair vehicle its coordinates. Since the "achieve" illocutionary force is used, this directly triggers the plan *!fix_car(X,Y)[source(car)]*: the helicopter leaves its parking to go to the broken down car.
  The movement toward the broken car is done comparing step by step its current position to the coordinates of the car, reaching first the same column, and then the right row.
  Once it gets there, it sends a message to the car to update its state percept back to "works" so that the car can move again. Then, the helicopter proceeds to head back towards its parking, which coordinates are observable in the environment.



```
[helicopter] join workspace /main/city: done
[helicopter] focusing on artifact view9 (at workspace /main/city) using namespace default
[helicopter] focus on view9: done
[helicopter] I'm in (9, 9)
[helicopter] I'm on my way to fix the car1 at (11;6)!
[helicopter] I'm in (10, 9)
[helicopter] Went right!
[helicopter] I'm in (11, 9)
[helicopter] Went right!
[helicopter] I'm in (11, 8)
[helicopter] Went up!
[helicopter] I'm in (11, 7)
[helicopter] Went up!
[helicopter] I'm in (11, 6)
[helicopter] Went up!
[helicopter] Just reached the car to repair in (11;6)! Car repair in progress...
[helicopter] car1 is fully repaired!
[helicopter] Going back to my parking in (9;9)
```

Figure 17: Logs of the helicopter fixing the car which called for help in Figure 9

- **Multiple cars call for help:**
  If a car broke down during the repairing of another car, or just after, the helicopter is able to take care of the second car before going back to its parking spot.
  This flexibility is necessary as the helicopter returning to the parking before being able to take into account any other SOS call would be a huge waste of time.
  To implement this behaviour, we used the *busy* belief: as soon as the helicopter receives a call, the busy belief becomes 1, and the plans to return to the parking cannot occur anymore. When the value is 0, the helicopter is free and can stay or return to the parking to wait for cars' calls.
  Moreover, if a car sends an SOS message to the helicopter while *busy(1)*, there is failure plan that waits a bit before calling again the *!fix_car(X, Y)[source(car)]* plan. This way, as soon as the helicopter is not busy anymore, so *busy(0)*, it can take into account the other car's call and head towards it to fix it.



```
[helicopter] I'm in (9, 9)
[helicopter] I'm on my way to fix the car3 at (5;3)!
[helicopter] I'm in (8, 9)
[helicopter] Went left!
[helicopter] I'm in (7, 9)
[helicopter] Went left!
[helicopter] I'm in (6, 9)
[helicopter] Went left!
[helicopter] I'm in (5, 9)
[helicopter] Went left!
[helicopter] I'm in (5, 8)
[helicopter] Went up!
[helicopter] I'm in (5, 7)
[helicopter] Went up!
[helicopter] I'm in (5, 6)
[helicopter] Went up!
[helicopter] I'm in (5, 5)
[helicopter] Went up!
[helicopter] I'm in (5, 4)
[helicopter] Went up!
[helicopter] I'm in (5, 3)
[helicopter] Went up!
[helicopter] Just reached the car to repair in (5;3)! Car repair in progress...
[helicopter] car4 also seems to need my help, but I'm busy for now. Call again later!
[helicopter] car3 is fully repaired!
[helicopter] Going back to my parking in (9;9)
[helicopter] I'm in (6, 3)
[helicopter] Went right!
[helicopter] I'm on my way to fix the car4 at (8;1)!
[helicopter] I'm in (7, 3)
[helicopter] Went right!
[helicopter] I'm in (8, 3)
```

Figure 18: Logs of the helicopter not going back to the parking spot because it received a SOS call from another car in the meantime

4 Implementation: Difficulties encountered

While implementing our system, we faced some hardships that made it mandatory for us to refine our design to cover these situations. Here is a list of the main topics we had to address, along with the technical solution we created to answer it:

- **Cars movement:**
  The plan *!change_direction* was going on in parallel to the main *!drive_random* and cars were behaved strangely. To avoid these interferences, we created the belief *busy*.
  When associated with a value of 0, the car is not moving and so it can follow the *!drive_random* plan. But when it is changing direction, the belief gets atomically updated to 1, and this regulates the movement by preventing the car to also execute its initial goal in parallel.

- **Cars going back forbidden:**
  A feature we wanted with cars' movement is to avoid going back when trying to change of direction. At the beginning, cars when trying to change of direction to avoid an obstacle, would use the *random_direction* custom internal action to draw a direction out of the four. This meant that it was possible for the car to choose the direction it was coming from, and hence to go back. This behaviour is wrong because it makes the car going the wrong way in its lane.
  To prevent this from happening, we hence created a subplan *!no_going_back* that would ensure that the new direction drawn was not one making the car going back. As arguments of this triggering event, we added the direction opposite to the last successful direction. This way, if a car performed a move and then encounters an obstacle making it having to change of direction, the forbidden direction will be the opposite move.

- **Agents exchanging behaviours:**
  As explained earlier, our agents move according to their types: a car uses the move function, a pedestrian the walk one, and an helicopter the fly one.
  The switch between these functions is taken care by the function move of City.java, the controller. In the first version of our code, this function was looking at the content of the cell, and there were three possible situations:

  - a cell with a car and no helicopter (then call for the move function);
  - a cell with a pedestrian and no helicopter (then call for the walk function);
  - a cell with the helicopter (then call for the fly function).

  The differentiation of these situations was based on the content of the cell rather then the type of agent calling for the move function.
  The problem is that if a helicopter and a car were on the same cell and called for the function move, there was a chance that the car would end up in the third situation, so starts flying. Because of this design defect, there were times where our helicopter started behaving like a car or a pedestrian and vice-versa, causing huge problems as the movement constrains of these vehicles are not the same.
  To correct this, we used the id of the agent calling the move to select the specific function. We created a list *agTypes* containing the type of the agent at the related position (for example, if the first agent to be instantiated in a car, agTypes[0]=CAR). Since the move controller function takes in argument the id of the agent calling, we could retrieve from that the type through our list, and hence trigger the adequate behaviour.

- **Pedestrian communication: names**
  In the synchronous act of communication between the pedestrians, the problem was that the ping agent has to know the name of the pong one.
  Having just the id of the neighbour pedestrian was not enough to communicate, as the proper agent's name was needed. To avoid having to hardcode it, we then defined some naming standards when instantiated pedestrians agents to the world:

⟨type of agent (car, or pedestrian_child, or pedestrian_adult, or helicopter)⟩
⟨n° of the instance (id)⟩

We later implemented several other things:

- a specific list for each type of agents: it contains the id of the agents of that type. This allowed us to be able to numerate the agents depending of their type instead of having to rely on their absolute id.
  Example: if ag0 is a car, ag1 is a car, ag2 is a child pedestrian, and ag3 is a car, we have agCars=[0,1,3] and agChildPedestrians=[2]
- a list agNames: it contains the name of each agent which id matches the specific index of the list.
  Example (follow-up of the previous example): agNames=[car1, car2, pedestrian_child1,car3]
- a function getAgNameAtPos(x,y): it retrieves the name of the agent given the id, using the agNames list.

With this way of handling things, we could add the name of the agent perceived in the fourth argument of the *who* percept, and this could perform adequate communication acts in the adult pedestrians playing the role of the ping.

- **Pedestrian communication: infinite greetings**
  While testing our pedestrian communication, we discovered that there were situations that could lead to the pedestrian staying still forever. The infinite communication was occurring because they greeted again the same agents over and over again.
  To avoid this, we created a belief *last_greeted* on the adult side: it gets updated at the very beginning of the communication act with the name of the child its speaking with. This enables to not greet the same agent twice in a row as the value of this *last_greeted* is then checked in the context of greetings triggering event.

- **Pedestrian communication: infinite waitings**
  Adults would sometimes perceive a child and send them a message, but the child would not receive it, leading the adult to stay stuck on the spot waiting infinitely for a greetings back that would never arrive. This situation most of all depends of the system's processing time, since percepts of surroundings are updated by the environment with a delay that can vary from one agent to another.
  To avoid this infinite waiting, we then created a plan *!wait_for_greetings_back* that acts as a stopwatch triggered as soon as the adult sent its greetings. This stopwatch performs three iterations, waiting for 800ms everytime before incrementing its belief count. If after three iterations the *greetings_back* percept has still not been received, the adult update its own belief waiting back to 0 and started again to move.
  Note that this belief waiting actually accomplish a similar role to the busy one for the cars.

- **Percepts in the belief base:**
  Our system relies on a strong and rather complex interaction between all the different types of agents and the environment. This latter hence manages a lot of percepts that it makes observable by every agents, which concretely means that they get integrated into their belief base.
  Cartago's framework eases the creation of such percepts thanks to some operations on properties. At first, we were using a function *addPercept* that was sending a percept (defined as a property) from the environment to the agents' belief base. However, this was making the belief bases overloaded and unreadable. Let's just think for example that every movement of every agent is followed by a success or fail percept, so the belief base was full of these, and with a lot of redundancy too. To prevent this situation from happening, we rather defined templates of percepts thanks to the Cartago *defineObsProperty* function. We then instantiated them with -1 or empty String, and we updated directly these inner arguments rather than sending a whole new percept in addition to the previous ones.

5 Deployment Instructions

In order to ease and automate the configuration and deployment of our system on different devices and different IDE, we wanted to use an automation tool. Our choice went to Gradle as our project is Java-based. In this way dependencies were managed automatically and the running instructions of this project are pretty easy:

1. Clone the repository from GitHub:

   ```
   git clone https://github.com/yourusername/MAS-
      CitySimulation.git

   cd MAS-CitySimulation
   ```

   or clone the repository from GitLab (you need access for it):

   ```
   git clone https://dvcs.apice.unibo.it/pika-lab/courses/
      mas/projects/mas-project-turrinideshays2324.git

   cd MAS-CitySimulation
   ```

2. Build the project with Gradle:

   ```
   ./gradlew build
   ```

3. Run the simulator:

   ```
   ./gradlew run
   ```

6 Conclusion

Through this project, we have been able to implement by ourselves a whole complex multi-agent system through a rather playful and motivating use case, the simulation of a city, which is clearly Bologna because of the orange buildings.
This was an enriching opportunity for us to better understand why agents are so special with their autonomous status, manipulating these concrete examples of the broader multi-agent theory we studied. We had the chance to get to know and even becoming more familiar with the programming of agents through Jason, and appreciate this language.
Working on a custom environment and on the visualisation through JaCaMo, and even managing to integrating the A Star algorithm to the whole picture, were interesting means to explore more in depth the behavioural patterns we had studied in lab sessions.

References

[1] *A Multiagents System of Gold Miners Game.* URL: `https://github.com/laraoberderfer/mas-gold-miners`.

[2] *A Star Pathfinding.* URL: `https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2`.

[3] *AgentSpeak Language.* URL: `https://en.wikipedia.org/wiki/AgentSpeak`.

[4] *BDI Architecture.* URL: `https://www.sciencedirect.com/topics/computer-science/belief-desire-intention-architecture`.

[5] *Cartago.* URL: `https://sourceforge.net/projects/cartago/`.

[6] *JaCaMo.* URL: `https://jacamo-lang.github.io/`.

[7] *Jason.* URL: `https://jason-lang.github.io/`.

[8] *Moise.* URL: `https://moise.sourceforge.net/`.