Taxi Simulator Documentation

Release 0.4.1

Javi Palanca

Jan 07, 2019

Contents

9   Indices and tables   63

Bibliography   65

Python Module Index   67

Contents:

Taxi Simulator

Agent-based taxi simulator to test strategies

- Free software: MIT license
- Documentation: https://taxi-simulator.readthedocs.io.

1.1 Features

- Strategy pattern
- Continuous simulator
- Load scenarios
- Multi-agent system built with SPADE
- XMPP communications

1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

Installation

2.1 Stable release

To install Taxi Simulator, run this command in your terminal:

```
$ pip install taxi_simulator
```

This is the preferred method to install Taxi Simulator, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Taxi Simulator can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/javipalanca/taxi_simulator
```

Or download the tarball:

```
$ curl  -OL https://github.com/javipalanca/taxi_simulator/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

Quickstart

Table of Contents

3.1 Usage

Using Taxi Simulator is as easy as running the application in a command line. There are two use modes: a command-line interface and a graphical web-based view. You can run simulations using only the command line or using the more easy and intuitive graphical user interface. Running Taxi Simulator without your own developed coordination strategies is posible since the application comes with a set of default strategies. Let's explore how to use both user interfaces.

3.2 Command-line interface

After installing Taxi Simulator open a command-line and type `taxi_simulator --host your_xmpp_server`. This starts a simulator without any options and runs the coordinator agent. The con-

sole will output the default logging information and you can terminate the simulator by pressing `Ctrl+C`. When you terminate the simulator the results of the simulations are printed.

> **Warning:** Note that `your_xmpp_server` is a fake address. You need to have an XMPP server where the simulator connects to. You can use your own XMPP server or use any of the public XMPP servers (List of public servers is *here <https://list.jabber.at>*).

Hint: To install an XMPP server visit https://xmpp.org/software/servers.html (we recommend Prosody IM)

```
$ taxi_simulator --host 127.0.0.1
INFO:root:Starting Taxi Simulator
INFO:CoordinatorAgent:Coordinator agent running
INFO:CoordinatorAgent:Web interface running at http://127.0.0.1:9000/app
INFO:root:Creating 0 taxis and 0 passengers.
INFO:RouteAgent:Route agent running
WARNING:RouteAgent:Could not load cache file.

^C
INFO:root: Terminating... (0.0 seconds elapsed)
Simulation Results

|   Avg Waiting Time |   Avg Total Time |   Simulation Time | Simulation Finished   |

|                  0 |                0 |                 0 | True                  |

Passenger stats

| name    | waiting_time   | total_time   | status   |


Taxi stats

| name    | assignments    | distance    | status    |

```

However, if you don't use some options when running the simulator there will be no default taxis nor passengers. That's why stats are empty. To run a simulation with some parameters you can use the command-line interface options.

To show these options you can enter the `--help` command:

```
$ taxi_simulator --help

Usage: taxi_simulator [OPTIONS]

  Console script for taxi_simulator.

Options:
  -n, --name TEXT             Name of the simulation execution.
  -o, --output TEXT           Filename to save simulation results.
  -of, --oformat [json|excel]  Output format used to save simulation
                              results. (default: json)
  -mt, --max-time INTEGER     Maximum simulation time (in seconds).
```

(continues on next page)

```
  -r, --autorun                 Run simulation as soon as the agents are
                                ready.
  -t, --taxi TEXT               Taxi strategy class (default:
                                AcceptAlwaysStrategyBehaviour).
  -p, --passenger TEXT          Passenger strategy class (default:
                                AcceptFirstRequestTaxiBehaviour).
  -c, --coordinator TEXT        Coordinator strategy class (default:
                                DelegateRequestTaxiBehaviour).
  --port INTEGER                Web interface port (default: 9000).
  -nt, --num-taxis INTEGER      Number of initial taxis to create (default:
                                0).
  -np, --num-passengers INTEGER Number of initial passengers to create
                                (default: 0).
  --scenario TEXT               Filename of JSON file with initial scenario
                                description.
  -cn, --coordinator-name TEXT  Coordinator agent name (default:
                                coordinator).
  --coord-passwd TEXT           Coordinator agent password (default:
                                coordinator_passwd).
  -rn, --route-name TEXT        Route agent name (default: route).
  --route-passwd TEXT           Route agent password (default: route_passwd).
  --host TEXT                   XMPP server address
  -ip, --ip-address TEXT        IP to serve web (default: 127.0.0.1).
  -v, --verbose                 Show verbose debug level: -v level 1, -vv
                                level 2, -vvv level 3, -vvvv level 4
  --help                        Show this message and exit.
```

3.2.1 Running a simulation from the command-line

To run a quick simulation from the command-line you need to set up a few arguments: the number of taxis, the number of passengers and (optionally) the maximum time of simulation. The argument `--num-taxis` (or `-nt`` initializes the specified number of taxis in random positions of the map. The argument `--num-passengers` (or `-np`` initializes the specified number of passengers in random positions of the map and with a random destination for each one. If you want limit the simulation time tou can use the `--max-time` argument (or `-mt`) to set the maximum number of seconds after which the simulation will end. Finally, the `--autorun` argument (or `-r`) automatically runs the simulation (this argument is important when you are not using the graphical interface, since it is the only way to start the simulation).

> **Warning:** The `--host` argument is important to specify where your XMPP server is (localhost or external)

Example:

```
$ taxi_simulator --host 127.0.0.1 --num-taxis 2 --num-passengers 2 --max-time 60 --
↪autorun
INFO:root:Starting Taxi Simulator
INFO:CoordinatorAgent:Coordinator agent running
INFO:CoordinatorAgent:Web interface running at http://127.0.0.1:9000/app
INFO:root:Creating 2 taxis and 2 passengers.
INFO:RouteAgent:Route agent running
INFO:CoordinatorAgent:Simulation started.
INFO:PassengerAgent:Passenger michelle08 asked for a taxi to [39.469057, -0.406452].
INFO:PassengerAgent:Passenger schapman asked for a taxi to [39.465762, -0.382746].
```

```
INFO:TaxiAgent:Taxi stevencortez sent proposal to passenger michelle08
INFO:TaxiAgent:Taxi austin05 sent proposal to passenger michelle08
INFO:PassengerAgent:Passenger michelle08 accepted proposal from taxi stevencortez@127.
→0.0.1
INFO:PassengerAgent:Passenger michelle08 refused proposal from taxi austin05@127.0.0.1
INFO:TaxiAgent:Taxi stevencortez on route to passenger michelle08
INFO:PassengerAgent:Passenger michelle08 informed of status: 11
INFO:PassengerAgent:Passenger michelle08 waiting for taxi.
INFO:TaxiAgent:Taxi stevencortez has arrived to destination.
INFO:PassengerAgent:Passenger schapman asked for a taxi to [39.465762, -0.382746].
INFO:TaxiAgent:Taxi austin05 sent proposal to passenger schapman
INFO:PassengerAgent:Passenger schapman accepted proposal from taxi austin05@127.0.0.1
INFO:TaxiAgent:Taxi austin05 on route to passenger schapman
INFO:PassengerAgent:Passenger schapman informed of status: 11
INFO:PassengerAgent:Passenger schapman waiting for taxi.
INFO:TaxiAgent:Taxi stevencortez has picked up the passenger michelle08@127.0.0.1.
INFO:PassengerAgent:Passenger michelle08 informed of status: 12
INFO:PassengerAgent:Passenger michelle08 in taxi.
INFO:TaxiAgent:Taxi stevencortez has arrived to destination.
INFO:TaxiAgent:Taxi stevencortez has dropped the passenger michelle08@127.0.0.1 in␣
→destination.
INFO:PassengerAgent:Passenger michelle08 informed of status: 22
INFO:PassengerAgent:Passenger michelle08 arrived to destination after 10.8725750446␣
→seconds.
INFO:TaxiAgent:Taxi austin05 has arrived to destination.
INFO:TaxiAgent:Taxi austin05 has picked up the passenger schapman@127.0.0.1.
INFO:PassengerAgent:Passenger schapman informed of status: 12
INFO:PassengerAgent:Passenger schapman in taxi.
INFO:TaxiAgent:Taxi austin05 has arrived to destination.
INFO:TaxiAgent:Taxi austin05 has dropped the passenger schapman@127.0.0.1 in␣
→destination.
INFO:PassengerAgent:Passenger schapman informed of status: 22
INFO:PassengerAgent:Passenger schapman arrived to destination after 22.221298933␣
→seconds.
INFO:root:
Terminating... (22.7 seconds elapsed)
INFO:CoordinatorAgent:Stopping taxi stevencortez
INFO:CoordinatorAgent:Stopping taxi austin05
INFO:CoordinatorAgent:Stopping passenger michelle08
INFO:CoordinatorAgent:Stopping passenger schapman
Simulation Results

| Avg Total Time  | Avg Waiting Time |  Simulation Time |  Max Time | Simulation␣
→Finished   |

|           22.69 |            16.55 |          22.6766 |        60 | True      ␣
→             |

Passenger stats

| name                 | total_time  | waiting_time | status           |

| michelle08@127.0.0.1 |      22.685 |      10.8726 | PASSENGER_IN_DEST |

| schapman@127.0.0.1   |     22.6845 |      22.2213 | PASSENGER_IN_DEST |

Taxi stats
```

```
 name                      |   assignments |   distance | status          |
 stevencortez@127.0.0.1 |              1 |     4835.1 | TAXI_WAITING  |
 austin05@127.0.0.1     |              1 |     7885.2 | TAXI_WAITING  |
```

By analyzing the output of the simulation we can see what events have occurred and how the simulation has been developed. There were created two taxis (`stevencortez` and `austin05`) and two passengers (`michelle08` and `schapman`). After the negotiation provided by the default strategies included in Taxi Simulator, taxi `stevencortez` was assigned to passenger `michelle08` and taxi `austin05` was assigned to passenger `schapman`. After 22 seconds of simulation both passengers were delivered in their destinations and taxis are free again to attend more passenger requests.

The output of the simulation also shows some statistics of the simulation, with the *Average Total Time*, which represents the average time of passengers from the moment they request a taxi until they are delivered to their destination, and the *Average Waiting Time*, which is the average time of passengers from requesting a taxi to being picked up. This information is also shown for each passenger along with their status at the end of the simulation.

In the case of taxis, the shown information includes the number of assignments of each taxi (how many passengers it has delivered), the total distance it has traveled and its final status.

This information is going to be useful for the development of new strategies that improve the system balancing or for debugging errors if a taxi or a passenger gets stuck or any other unexpected situation occurs.

3.2.2 Saving the simulation results

If you want to store the results of simulation in a file you may use the `--output` option (or `-o`) to specify the name of the file where the simulation results will be saved. The `--oformat` (`-of`) allows you to choose the output format between json (default) or excel. It is also useful to use the `--name` (or `-n`) to name the simulation.

Example:

```
$ taxi_simulator --host 127.0.0.1 --name "My Simulation" --output results.xls --
→oformat excel
```

3.2.3 Advanced options

There are other options that are less common and that you probably don't need to use very often. These are options that allow you to change connection ports or default name and password of the coordinator agent. Use them only if there is a port or name conflict.

The last but no less important option is the verbosity option. It allows you to specify how verbose you want the simulator to be. The number of `v` letters you pass to the option indicates the level of verbosity (e.g. `-v` is **DEBUG** verbosity and `-vvvv` is the highest level of verbosity where the internal messages of the platform are shown).

Note: You may have noticed that we haven't discussed three very important options that are: `--taxi`, `--passenger`, and `--coordinator`. These options are used to inject new strategies to the simulator and we'll be discussed in a later chapter. Also, the `--scenario` option will be fully explained in a later section.

3.3 Graphical User Interface

A much more user-friendly way to use Taxi Simulator is through the built-in graphical user interface. This interface is accessed via any web browser. Just look at the address shown on the screen when you run the simulator and access that website.

Hint: The Coordinator agent is who raises the GUI and shows the address in the debug:

```
INFO:CoordinatorAgent:Web interface running at http://127.0.0.1:9000/app
```

This address is (in most cases): http://127.0.0.1:9000/app

Once you visit the GUI address you see an interface like this:
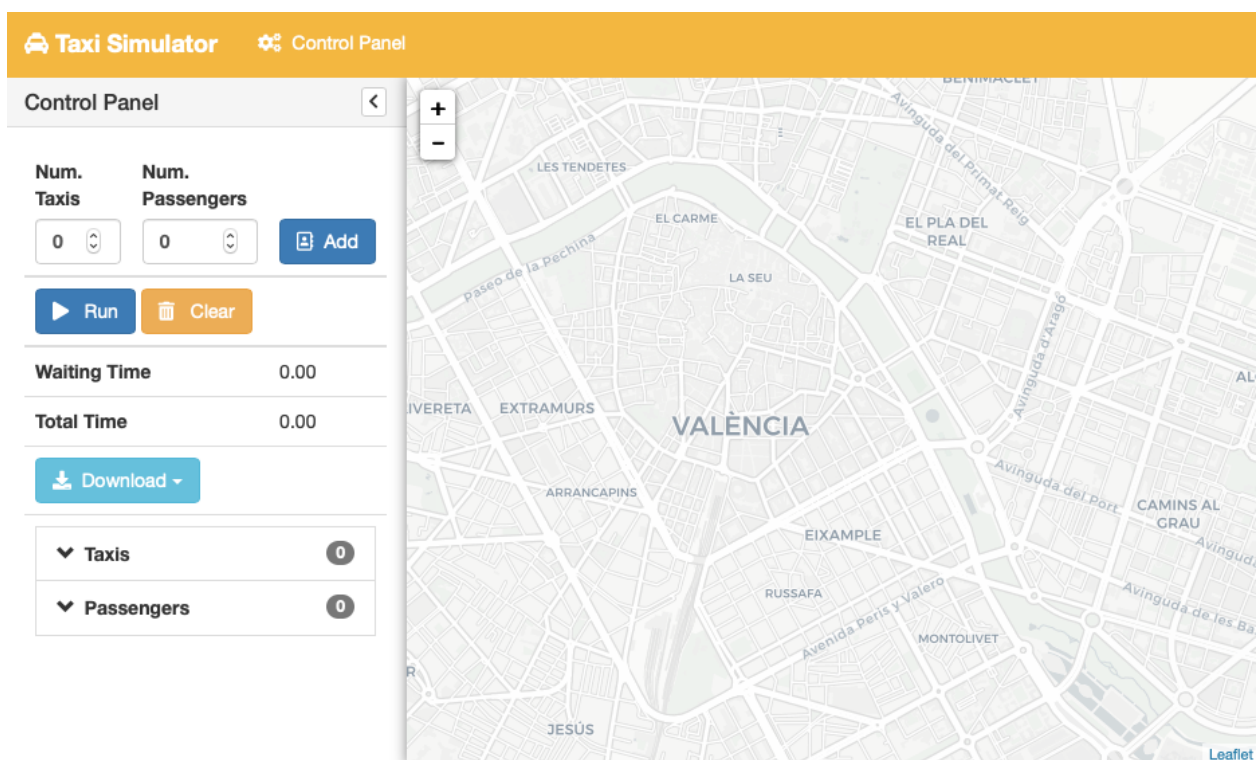
Fig. 1: GUI at startup

In the GUI you can see a map of the city on the right and a Control Panel with various options on the left:

1. Two selectors to set the number of taxis and passengers and an **Add** button. When this button is pressed the number of taxis and passengers that are in the input boxes are created in random positions inside the map. This form is very similar to the command line option, except that you can add Taxi and Passenger agents at any time during the simulation.

2. A **Run** button that starts the simulation.

3. A **Clear** button to stop and reset the simulation.

4. Stats of the waiting time and total time of the simulation in real time.

5. A **Download** button to get the stats of the simulation in excel or json format.

6. A collapsable tree view with the taxis and passengers that are included in the simulation, with a color bullet that indicates their current status.

If the **Run** buttons is pressed the simulation shows how the taxis move to the passengers and deliver them to their destinations.
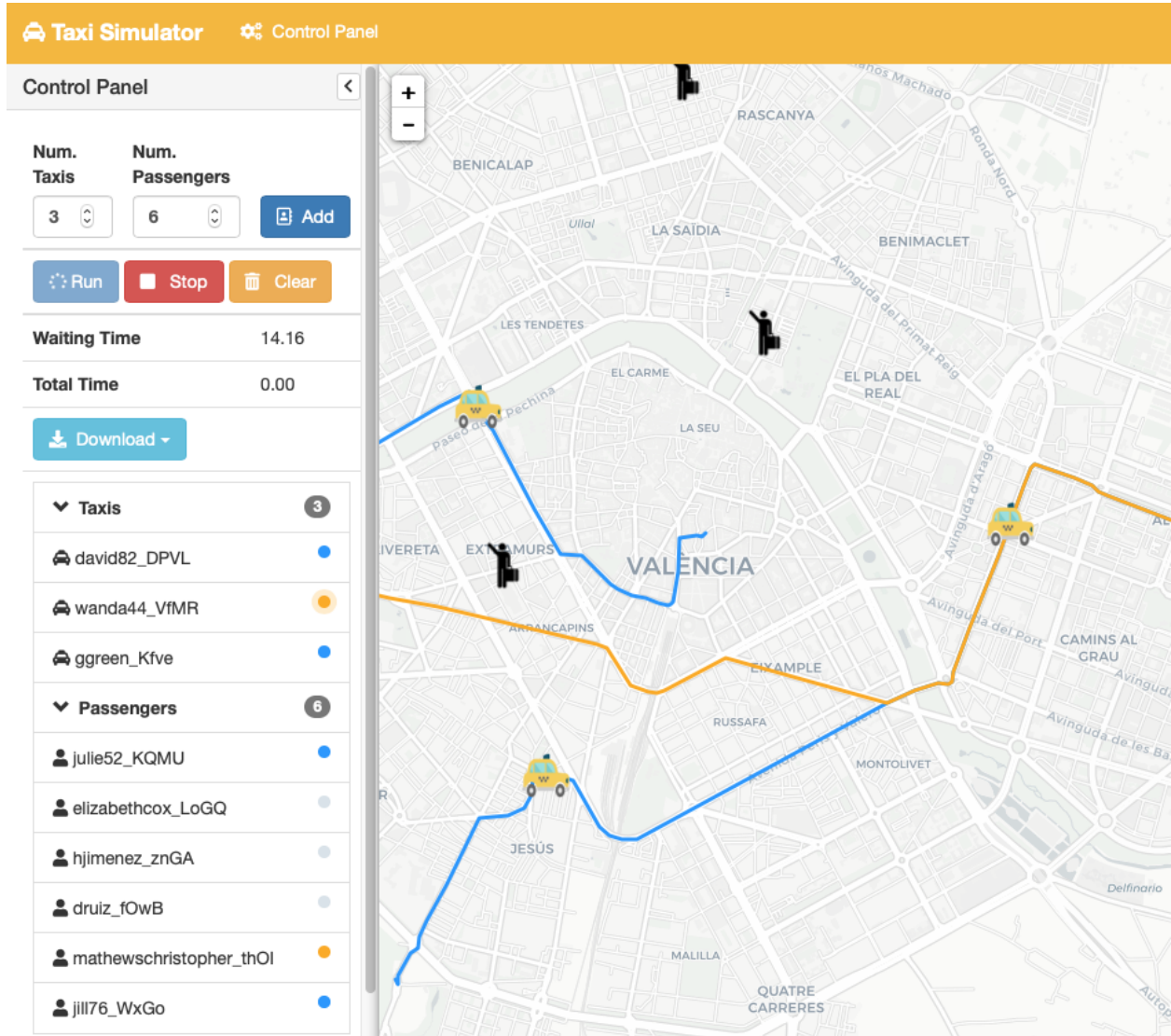

Fig. 2: Simulation in progress

Notice that when a taxi picks up a passenger, the passenger's icon disappears from the map view (since it is inside the taxi) and is no longer viewed (it's also not shown when it arrives to its desination). However, you can check at any time your passengers status in the tree view of the Control Panel.

The code colors in the tree view indicate the status of a taxi or a passenger. The legend of colors is as follows:

Taxis		Passengers	
Bullet	Status	Bullet	Status
🟢	WAITING	🔵	WAITING
🟠	WAITING FOR APPROVAL	🟠	ASSIGNED
🟠	MOVING TO PASSENGER	🔵	IN TAXI
🔵	MOVING TO DESTINATION	🟢	IN DESTINATION

Hint: Every time than a bullet is pulsing means that the agent is moving.

When a taxi is moving it's also shown in the GUI the path that the taxi is folowing. The color of the path indicates the type of movement than the taxi is doing. A yellow path indicates that the taxi is going to pick up the passenger. On the other hand, a blue path indicates that the taxi is taking the passenger to his destination.

Note: A simulation is finished when all taxis are free (and waiting for new passengers) and all passengers are in their destinations (i.e. all bullets are green).

3.4 Loading Scenarios

Adding agents using both the graphical interface and command line is convenient and fast, but if you want to perform repeatable experiments where you choose where agents appear and what the destinations of the passengers are (rather than random data) then you need the mechanism of the **scenarios**.

The ability to load scenarios to Taxi Simulator allows us to repeat the same experiment as many times as we want with the same initial conditions. Taxi Simulator supports to load a *scenario* file that defines all the fields that you need to load the same information repeatedly. A scenario file must be coded in JSON format.

The fields that the scenario file must include are a passengers list and a taxis list. Each passenger must include the following fields:

Field	Description
position	Initial coordinates of the passenger
dest	Destination coordinates of the passenger
name	Name of the passenger
password	Password for registering the passenger in the platform (optional)

For taxis the fields are as follows:

Field	Description
position	Initial coordinates of the taxi
name	Name of the taxi
password	Password for registering the taxi in the platform (optional)
speed	Speed of the taxi (in meters per second)

An example of a scenario file with two passengers and two taxis:

```json
{
    "passengers": [
        {
            "dest": [ 39.463356, -0.376463 ],
            "position": [ 39.460568, -0.352529 ],
            "name": "michaelstewart",
            "password": "T3TnmjuI(m"
        },
        {
            "dest": [ 39.49529, -0.401478 ],
            "position": [ 39.49529, -0.401478 ],
            "name": "ghiggins",
            "password": "@5wPA$Mx#O"
        }
    ],
    "taxis": [
        {
            "position": [ 39.462618, -0.364888 ],
            "name": "taxi1",
            "password": "$JM!Zcwh0R",
            "speed": 2000
        },
        {
            "position": [ 39.478458, -0.406736 ],
            "password": "_bx1TBEiu8",
            "name": "taxi2",
            "speed": 2000
        }
    ]
}
```

Finally, to load a scenario in a simulation use the `--scenario` option with the filename of the JSON file:

```
$ taxi_simulator --host 127.0.0.1 --scenario my_scenario.json

INFO:root:Starting Taxi Simulator
INFO:CoordinatorAgent:Coordinator agent running
INFO:CoordinatorAgent:Web interface running at http://127.0.0.1:9000/app
INFO:root:Creating 0 taxis and 0 passengers.
INFO:root:Loading scenario my_scenario.json
INFO:RouteAgent:Route agent running
```

Developing New Strategies

Table of Contents

4.1 Introduction

One of the main features of "Taxi Simulator" is the ability to change the default negotiation strategy of the agents that interact during the simulation: the Coordinator agent, the Taxi agents and the Passenger agents. The overall goal of the negotiation strategy of these three agent types is to decide which Taxi agent will transport each Passenger agent to its destination, making sure that no Passenger agent is left unattended. Additionally, the negotiation strategy may also try to optimize some metrics, such as the average time that Passenger agents are waiting to be served, or that the amount of gas spent by Taxi in their movements.

The negotiation strategy is based on two main elements. First, it is based on the internal logic of each agent type (Coordinator, Taxi and Passenger) and, in particular, on their respective *strategy behavior*, which includes the internal logic of each agent type regarding the negotiation process. And second, it is also based on the so-called *REQUEST* protocol, which comprises the types of messages exchanged among the three agent types during the negotiation. The following diagram presents the protocol in the typical FIPA format, where agents types are depicted as vertical lines and the exchanged message types (or "performatives") in horizontal arrows:
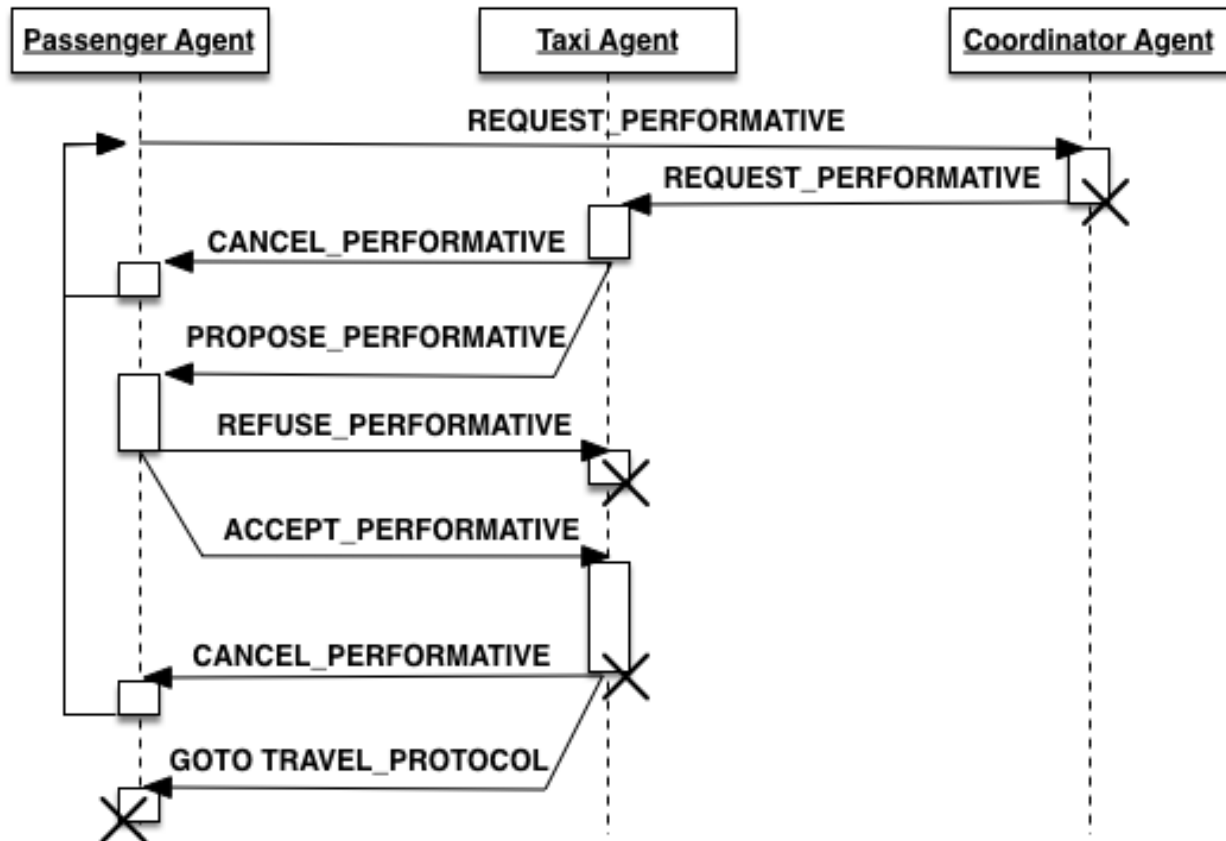
This chapter introduces first the current, default strategy of each agent type (Coordinator, Taxi and Passenger) and then explains how to introduce new strategies for any, or all, of them.

4.1.1 Description of the Coordinator Agent

The Coordinator Agent is responsible for putting in contact the Passenger agents that need a taxi service, and the Taxi agents that may be available to offer these services. In short, the Coordinator Agent acts like a taxi call center, accepting the incoming requests from customers (Passenger agents) and forwarding these requests to the (appropriate) Taxi agents. In order to do so, the Coordinator agent knows the names and addresses of every Passenger and Taxi agent registered in the system.

In the context of the Taxi Simulator, a "taxi service" involves, once a particular Passenger and Taxi agents have reached an agreement, the movement of the Taxi agent from its current position to the Passenger's position in order to pick the Passenger up, and then the transportation of the Passenger agent to its destination.

The Coordinator Agent includes a single behavior, which is its strategy behavior, now described.

Strategy Behaviour (*DelegateRequestTaxiBehaviour*)

The goal of the stategy behavior of the Coordinator Agent is basically to **receive** the "request" messages (*RE-QUEST_PERFORMATIVE*) sent by the Passenger agents that need a taxi service and, for each request, selecting the Taxi agent, or agents, that may perform the service, and **forward** the request to them. A *REQUEST_PERFORMATIVE* message includes the following fields:

```
"passenger_id": Id of the Passenger agent that performs the request.
"origin":       Current position of the Passenger, where the Taxi has to pick it up.
"dest":         Destination of the Passenger, where the Taxi needs to transport it.
```

The particular set of Taxi agents to which the request will be forwarded depends on the *allocation policy* of the Coordinator Agent, which is part of the strategy. In the default strategy behavior for the Coordinator agent (*DelegateRequestTaxiBehaviour*), the allocation policy is the simplest posible: it forwards every incoming request to **all** the Taxi agents, regardless of their current statuses or any other consideration (such as, for example, the last time they performed a service, or the distance between them and the Passenger agent).

In the default strategy behavior, the set of incoming messages that may be delivered to the Coordinator Agent is reduced to the requests made by Passenger agents, and the behavior itself does not include multiple states. So, each incoming message is processed in the same way, and leaves the behavior in the same (unique) state.

Once each request has been forwarded to some (or all) the Taxi agents, the goal of the Coordinator Agent for that request is achieved. This is the starting point to the negotiation between the Passenger that has issued the request and the Taxi agents that have received it, which is described in the following sections.

4.1.2 Description of the Taxi Agents

The Taxi agents represent vehicles which can transport Passenger agents from their current positions to their respective destinations. In order to do that, Taxi agents incorporate two behaviors: the strategy behavior and the moving behavior, now described.

Strategy Behaviour (*AcceptAlwaysStrategyBehaviour*)

The goal of the strategy behavior of a Taxi agent is to negotiate with Passenger agents which are requesting a taxi service the conditions of the service offered by the Taxi, in order to achieve an agreement with these Passenger agents. When an agreement is reached between a particular Passenger and Taxi agents, then the Taxi agent picks up the Passenger agent and transport it to its destination (and starts the Moving Behavior, described below).

The currently implemented, default strategy behavior is called *AcceptAlwaysStrategyBehaviour*, and has a direct relation with the *REQUEST* protocol explained above. In particular, the behavior can be thought of as a finite-state machine with some different states specifying the statuses of the Taxi agent regarding the strategy behavior, and some transitions between states, wich are triggered either by messages (of the *REQUEST* protocol) received by the Taxi agent, or by some other program conditions. This is depicted in the following diagram:

The semantics of each state are now described:

- *TAXI_WAITING*: In this state, the Taxi agent is available (free) and waiting for requests from Passenger agents. While in this state, if it receives a request message (*REQUEST_PERFORMATIVE*) from a particular Passenger agent, it will send the Passenger a service proposal (*PROPOSE_PERFORMATIVE*) and it will change its state to *TAXI_WAITING_FOR_APPROVAL*.

- *TAXI_WAITING_FOR_APPROVAL*: In this state, the Taxi agent is waiting for the response message from a Passenger agent to which it has sent a service proposal message. While in this state, it may receive two alternative answers from the Passenger agent: (1) the Passenger refuses the service proposal (*REFUSE_PERFORMATIVE*), in which case the Taxi changes its state back to *TAXI_WAITING*; or (2) the Passenger accepts the proposal (*ACCEPT_PERFORMATIVE*), in which case it will change to the state *TAXI_MOVING_TO_PASSENGER*.

- *TAXI_MOVING_TO_PASSENGER*: In this state, the Taxi agent and the Passenger agent have agreed to perform a taxi service, and then the Taxi agent starts to travel to the Passenger location in order to pick it up. This is the final state of the negotiation between the Taxi and a certain Passenger agent. In this state, the Taxi agent executes the helper function *pick_up_passenger*, which automatically starts the so-called Moving Behavior in the Taxi agent, described below. It also sends a message to the Travel Behavior of the Passenger agent, which starts that behavior (this is explained in the next section).
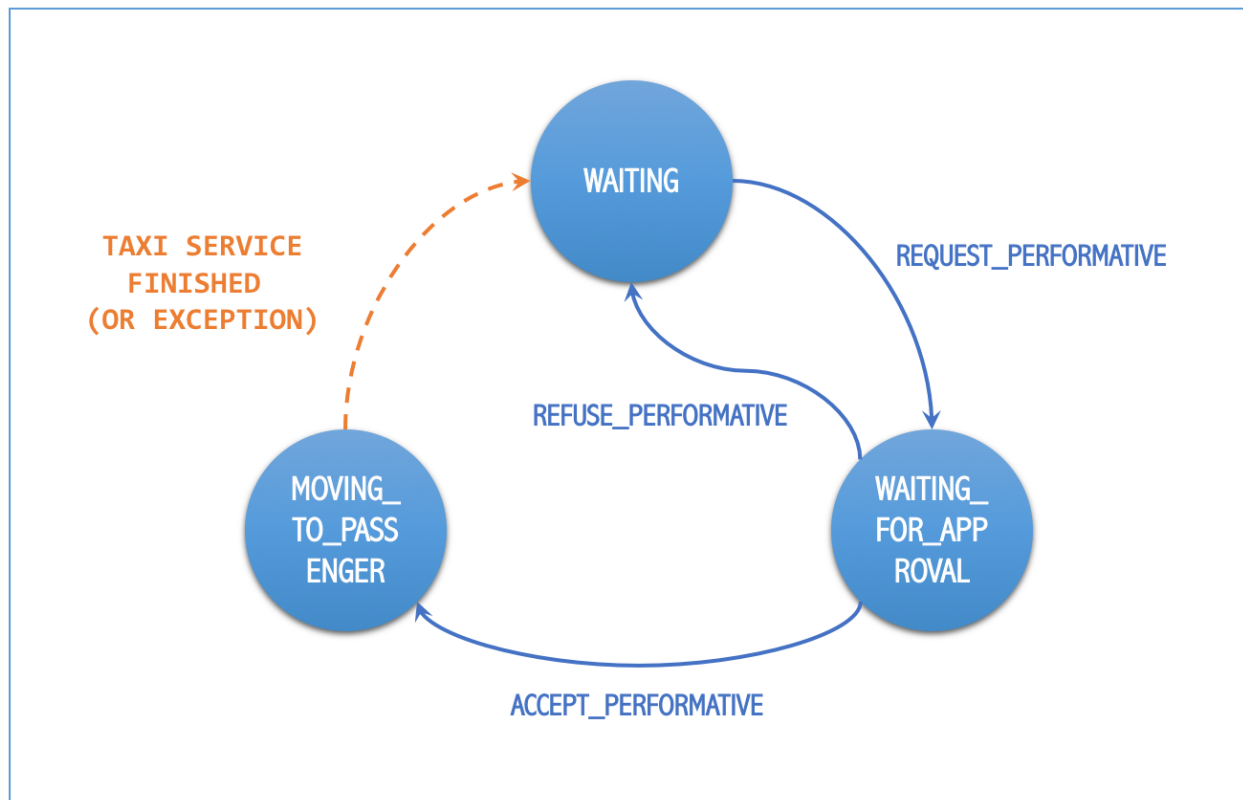
Fig. 1: States and transitions of the strategy behavior of a Taxi agent.

Moving Behaviour

This behavior makes the Taxi agent to move to the current location of the Passenger agent with which it has reached an agreement to perform a taxi service. After picking the Passenger agent up, the Taxi will then transport it to its destination. During that travel, the behavior informs the Passenger agent of where the Taxi is and what it is doing (going to pick up the Passenger, taking the Passenger to its destination, reaching the destination, etc.). All this is performed by sending the Passenger agent some messages which belong of another, dedicated protocol called *TRAVEL_PROTOCOL*.

Once the Taxi reaches the Passenger agent's destination and the Passenger agent is informed about it, the state of the Taxi agent (of the strategy behavior) is here changed to *TAXI_WAITING*, indicating that it is now free, and hence making the Taxi agent available again to receiving new requests from other Passenger agents.

> **Warning:** This behavior is internal and automatic, and it is not intended to be modified while developing new negotiation strategies. The same applies to the *TRAVEL_PROTOCOL* protocol.

4.1.3 Description of the Passenger Agents

The Passenger agents represent people that need to go from one location of the city (their "current location") to another (their "destination"), and for doing so, they request a taxi service. Each Passenger agent requires a single taxi service and so, once transported to its destination, it reaches its final state and ends its execution. During that execution, Passenger agents incorporate two behaviors: the strategy behavior and the travel behavior, now described.

Strategy Behaviour

In the course of the *REQUEST* protocol, the request of a taxi service made by a Passenger agent is answered by one (or several) Taxi agents, each of which offering the Passenger their conditions to perform such service. The goal of the strategy behavior of a Passenger agent is to select the best of these taxi service proposals, according to its needs and/or preferences (e.g., to be picked up faster, to get the nearest available taxi, to get the cheapest service, etc.).

The currently implemented, default strategy behavior is called *AcceptFirstRequestTaxiBehaviour*. As in the strategy behavior of the Taxi agents above, here we can also consider the strategy as a finite-state machine related to the messages (of the *REQUEST* protocol) received by the Passenger agent, as depicted below:

The semantics of each state are now described:

- *PASSENGER_WAITING*: In this state, the Passenger agent requires a taxi service and, periodically, sends a request for that service until one (or many) Taxi agent proposals (*PROPOSE_PERFORMATIVE*) are received. When the Passenger accepts a particular proposal (in the current implementation, always the first one it receives while in this state) then it communicates so to the proposing Taxi agent, and changes its own status to *PASSENGER_ASSIGNED*.

- *PASSENGER_ASSIGNED*: In this state, the Passenger agent has been assigned to a particular taxi, and the taxi service is being produced. The Passenger side of the taxi service is implemented by activating the Travel Behavior, described below, which is started by a message sent by the Taxi agent (in its helper function *pick_up_passenger*). If something goes wrong (for example, an exception is raised during the taxi service) or the Taxi agent voluntarily wants to cancel the service, then the Taxi agent sends a *CANCEL_PERFORMATIVE* to the Passenger agent, which would then change its status back to *PASSENGER_WAITING*, initiating the request process again.
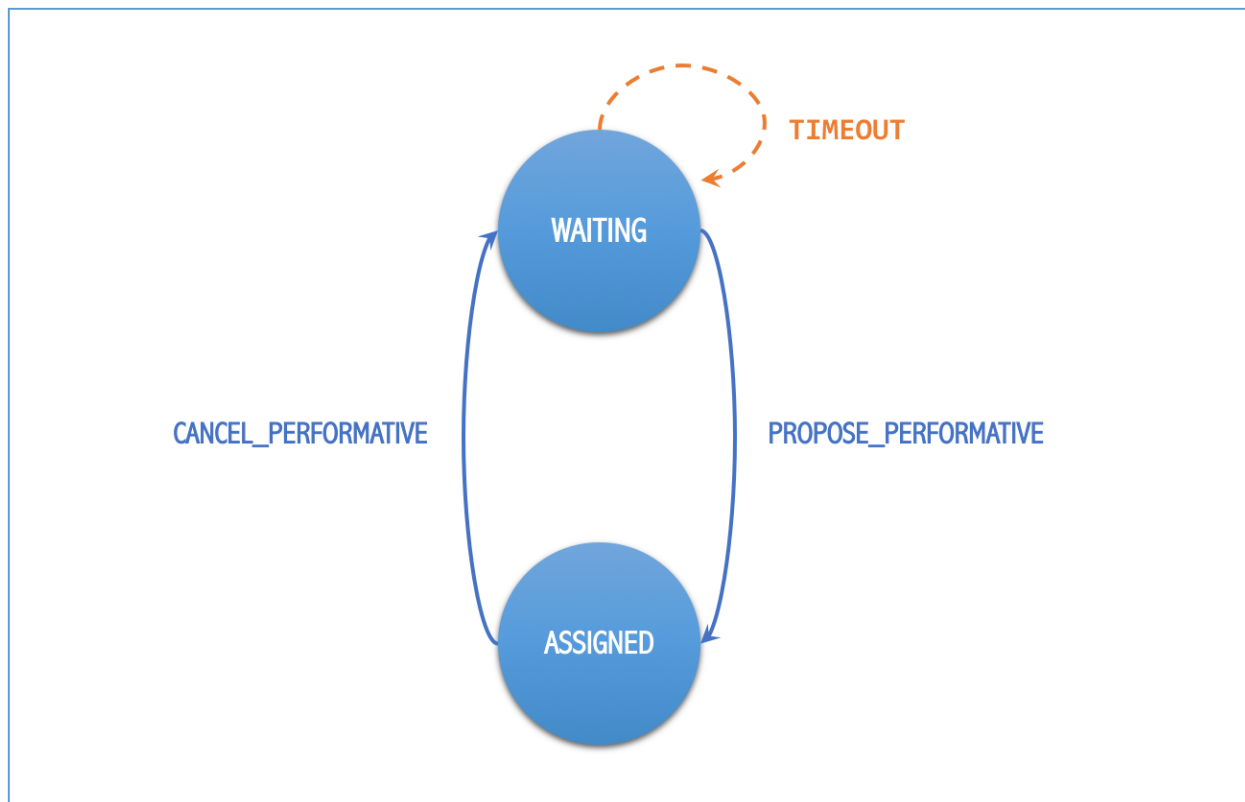
Fig. 2: States and transitions of the strategy behavior of a Passenger agent.

Travel Behaviour

This behavior is activated (in the Passenger agent) when a Taxi agent decides to pick up the Passenger agent, by means of a message sent by the Taxi (inside the Taxi agent's helper function *pick_up_passenger*). This message, as well as other messages sent by the Taxi agent to this behavior, belongs to a protocol called the *TRAVEL_PROTOCOL*.

The messages of the *TRAVEL_PROTOCOL* drive the transitions between the different states of this behavior, in the same way that the *REQUEST_PROTOCOL* does for the strategy behavior. In particular, the states of this behavior are: *PASSENGER_IN_TAXI*, when the Taxi agent has reached the Passenger agent's position and has picked it up; and *PASSENGER_IN_DEST*, when the Taxi agent has reached the Passenger agent's destination. This would be the final state of the Passenger agent.

> **Warning:** This behavior is internal and automatic, and it is not intended to be modified while developing new negotiation strategies. The same applies to the *TRAVEL_PROTOCOL* protocol.

4.1.4 The Negotiation Process between Taxi and Passenger Agents

After separately explaining the strategy behavior of Taxi and Passenger agents, this section tries to relate both behaviors. This is important to understand how these two agent types interact with each other in order to coordinate and reach the overall goals of the simulation.

In particular, there are three key aspects (embedded within the strategy behaviors) which influence the overall coordination process implemented in the simulator, as now described:

- The conditions of a taxi service proposal. The current implementation does not consider any special condition other than the Taxi agent being free (available to perform the service). Some aspects that could be included in a taxi proposal would be, for example, the current location of the taxi, the proposed fare, the route to take the Passenger agent to its destination, etc.

- The preferences of passengers in order to select a particular taxi proposal. In the current implementation, the Passenger agents always accept the first proposal received from a Taxi agent. In a more sophisticated negotiation, some internal goals/conditions of the Passenger agent could be taken into account in order to select a "better" proposal. These might include, for example, the expected waiting time until the Taxi agent arrives, the amount of money that the service is expected to cost, the brand of the Taxi vehicle, etc.

- The possibility of a taxi to voluntarily cancel an ongoing taxi service after a proposal has been accepted by a passenger. This may happen only before the pasenger has been picked up, that is, while the taxi is moving from its initial position to the location where the passenger is waiting for it. In the current implementation, a taxi service cancellation can only be produced if some exception is raised while the service is being produced (for example, if the software calculating a route for the Taxi agent fails to produce a valid route). Since new Passenger (and maybe Taxi) agents can appear at any time while the simulation is running, a voluntary cancellation of taxi services could improve the overall transportation of passengers throughout the simulation, allowing for a "dynamic reallocation" of passengers to taxis, even when taxi services where already committed.

4.2 Agent Foundations

The architecture of Taxi Simulator is built on top of a multi-agent system platform called SPADE. Although it is not necessary to build new agents in order to develop new coordination strategies (the simulator provides all the necessary agents), it is interesting to know how they work and what methods they provide for the creation of coordination strategies.

Next we will present the SPADE platform and its main features. For more documentation you can visit their website https://github.com/javipalanca/spade.

4.2.1 SPADE

SPADE (Smart Python multi-Agent Development Environment) is a multi-agent system (MAS) platform based on the XMPP technology and written in the Python programming language. This technology offers by itself many features and facilities that ease the construction of MAS, such as an existing communication channel, the concepts of users (agents) and servers (platforms) and an extensible communication protocol based on XML.

Extensible Messaging and Presence Protocol (XMPP) is an open, XML-inspired protocol for near-real-time, extensible instant messaging (IM) and presence information. The protocol is built to be open and free, asynchronous, decentralized, secure, extensible and flexible. The latter two features allow XMPP not only to be an instant messaging protocol, but also to be extended and used for many tasks and situations (IoT, WebRTC, social, . . .). SPADE itself uses some XMPP extensions to provide extended features to its agents, such as remote procedure calls between agents (Jabber-RPC), file transfer (In-Band Bytestreams), an so on.

In order to fully understand how SPADE works, it is necessary to know how the agents are made up and how they communicate. In the following sections we will summarize the SPADE agent model and its communication API.

Agent Model: Behaviors

SPADE agents are threaded-based objects that can be run concurrently and that are connected to a SPADE platform, which internally runs an XMPP server. Each agent must provide an ID and password in order to be allowed to connect to the platform. The agent ID is called JID and has the form of an email: a user name string plus a "@" character plus the IP address of the SPADE server to connect to (e.g. *my_agent@127.0.0.1*).

The internal components of the SPADE agents that provide their intelligence are the **Behaviors**. A behavior is a task that an agent can run using some pre-defined repeating pattern. For example, the most basic behavior type (pattern) is the so-called cyclic behavior, which repeatedly executes the same method over and over again, indefinitely. This is the way to develop typical behaviors that wait for a perception, reason about it and finally execute an action, and then wait for the next perception.

The following example is a sample of an agent with a cyclic behavior (`spade.behaviour.CyclicBehaviour` type) that waits for a perception from the keyboard input, reasons on it and executes an action, and continues to do so indefinitely until the user presses Ctrl+C. In order to build a behavior, you need to inherit from the type of behavior you want (in the case of this example, the cyclic behaviour is implemented in the class `spade.behaviour.CyclicBehaviour`) and overload the coroutine `run` where the body of the behavior is implemented. If needed, you can also overload the `on_start` and `on_end` coroutines in order to execute actions on the initialization or shutdown of a behavior, respectively.

```python
import spade
import datetime
import time

class MyAgent(spade.agent.Agent):
    class MyBehaviour(spade.behaviour.CyclicBehaviour):

        async def on_start(self):
            print("Initialization of behavior")

        async def run(self):
            # wait for perception, raw_input is a blocking call
            perception = raw_input("What's your birthday year?")
            # reason about the perception
            age = datetime.datetime.now().year - perception
            # execute an action
            print("You are {age} years old.".format(age=age))
```

(continues on next page)

```python
        async def on_end(self):
            print("Shutdown of behavior")

    def setup(self):
        # Create behavior
        behaviour = self.MyBehaviour()
        # Register behavior in agent
        self.add_behaviour(behaviour)

if __name__ == "__main__":
    a = MyAgent(jid="agent@127.0.0.1", password="secret")
    a.start()
    while True:
        try:
            time.sleep(1)
        except KeyboardInterrupt:
            break
    a.stop()
```

Along with the cyclic repeating pattern (or type), SPADE also provides several other types of behaviors, such as like one-shot behaviors, periodic behaviors, finite-state machine behaviors, etc. It is important to note that SPADE agents can execute many behaviors simultaneously, from the same or different types.

Communication API, Messages and Templates

Communication is one of the cornerstones of any multi-agent system, and SPADE is no exception. Agents can send and receive messages using a simple API, and more importantly, they can receive them in certain behaviors according to templates they can define.

A `spade.message.Message` is the class that needs to be filled in order to send a message. A Message may be filled with several pieces of information, but the most important fields are the receiver, the content, the performative and the protocol. The receiver must be filled with a *jid* address , which is a string. The content is the (string-based) body of the message. The performative and protocol both add semantic information to the message in the context of a conversation: they are normally used to represent the action and the rules that determine how the agents are going to communicate in a specific semantic context and they are represented as metadata.

Tip: It is usually recommended to use a representation language for the content of the message. Although semantic languages like OWL or RDF are normally used for this purpose, in this simulator JSON is used instead, for the sake of simplicity.

All these fields have a getter and setter function. An example is shown next:

```python
import spade

msg = spade.message.Message()
msg.to = "receiver_agent@127.0.0.1"
msg.set_metadata("performative", "request")
msg.set_metadata("protocol", "my_custom_protocol")
msg.body = "{'a_key': 'a_value'}"
```

Hint: Other metadata fields that can be filled in the message are the content language, the ontology, and so on.

The next step is to send the message. This is done with the `send` coroutine provided by a *Behaviour*. For example:

```python
import spade


class SenderAgent(spade.agent.Agent):
    class SendBehav(spade.behaviour.OneShotBehaviour):

        async def run(self):
            msg = spade.message.Message()
            msg.to = "receiver@127.0.0.1"
            msg.set_metadata("performative", "inform")
            msg.set_metadata("ontology", "myOntology")
            msg.set_metadata("language", "OWL-S")
            msg.body = "Hello World"

            await self.send(msg)  # send the message

    def setup(self):
        print "MyAgent starting..."
        behav = self.SendBehav()
        self.add_behaviour(behav)
```

The reception of messages is particular in SPADE, since messages can only be received by behaviors, and so SPADE provides each behavior executed by any agent with its own mailbox, and defines a mechanism in order to configure the particular behavior that must receive each message, according to the message type. This mechanism is carried out with *Templates*. When an agent receives a new message it checks if the message matches each of the behaviors using a template with which they where registered. If there is a match, the message is delivered to the mailbox of the corresponding behavior, and will be read when the behavior executes the `receive` method. Otherwise, the message will be dropped.

Note: The `receive` coroutine accepts an optional parameter: **timeout=seconds**, which allows the coroutine to be blocking until the specified number of seconds have elapsed. If the timeout is reached without a message being received, then `None` is returned. If the timeout is set to 0, then the `receive()` function is non-blocking and (immediately) returns either a `spade.message.Message` or `None`.

A `spade.template.Template` is created using the same API of `spade.message.Message`:

```python
import spade
template = spade.template.Template()
template.set_metadata("ontology", "myOntology")
```

Note: A `spade.template.Template` accepts boolean operators to combine *Templates* (e.g. `my_tpl = Template( template1 & template2))`

At this point we can present a full example on how to build an agent that registers a behavior with a template and receives messages that match that template:

```python
import spade
import asyncio


class RecvAgent(spade.agent.Agent):
    class ReceiveBehav(spade.behaviour.CyclicBehaviour):
```

```python
    async def run(self):
        await msg = self.receive(timeout=10)

        # Check wether the message arrived
        if msg is not None:
            assert "myOntology" == msg.get_metadata("ontology")
            print("I got a message with the ontology 'myOntology'")
        else:
            print("I waited 10 seconds but got no message")

def setup(self):
    recv_behav = self.ReceiveBehav()
    template = spade.template.Template()
    template.set_metadata("ontology", "myOntology")

    self.add_behaviour(recv_behav, template)
```

These are the basics of SPADE programming. You will not need to create all these structures, templates and classes in order to use *Taxi Simulator*, but it is always better to know the foundations before getting down to business.

4.3 How to Implement your own Strategies

Taxi simulator is designed for students to implement and test new strategies that lead to system optimization. The goal of this educational simulator is to make it easier for students to work with new coordination strategies without having to introduce major modifications to the application. For this purpose, Taxi Simulator incorporates the so-called Strategy design pattern, which is now introduced.

4.3.1 The Strategy Pattern

The **Strategy Pattern** is a design pattern that enables selecting an algorithm at runtime. The Strategy Pattern is the best practice when an application incorporates different, alternative versions of an algorithm and we want to be able to select any of these versions to be executed at run time. With this pattern, you can define a separate strategy (implementation of the algorithm) in an object that encapsulates the algorithm. The application that executes the algorithm **must** define an interface that every strategy (implementation) will follow, as it can be seen in the following figure:

Following this implementation, the context object can call the current strategy implementation without knowing how the algorithm was implemented. This design pattern was created, among others, by a group of authors commonly known as the **Gang of Four** (E. Gamma, R. Helm, R. Johnson and J. Vlissides), and it is well presented in *[GangOfFour95]*.

Taxi Simulator uses the *Strategy Pattern* in order to enable students to implement three different strategies (one for the coordinator agent, one for the taxi agent and one for the passenger agent) without having to develop new agents or entering in the complexity of the simulator. Thanks to this pattern, students can develop their strategies in an external file and pass it as an argument when the simulator is run.

Taxi Simulator implements one interface for each of these three agents, with each interface also providing some helper functions that intend to facilitate the most common actions of each (subclassed) agent. These three interfaces inherit from the `StrategyBehaviour` class and are called: `CoordinatorStrategyBehaviour`, `TaxiStrategyBehaviour` and `PassengerStrategyBehaviour`.
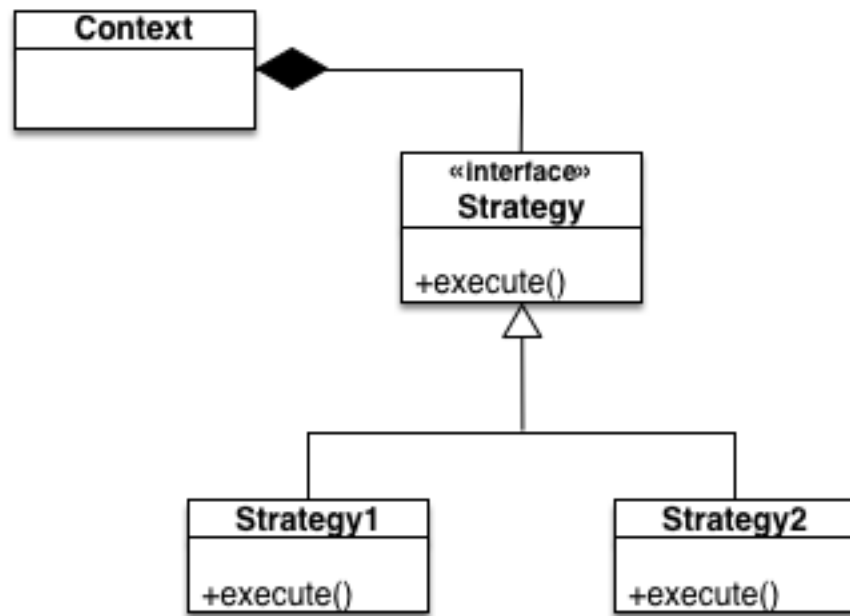
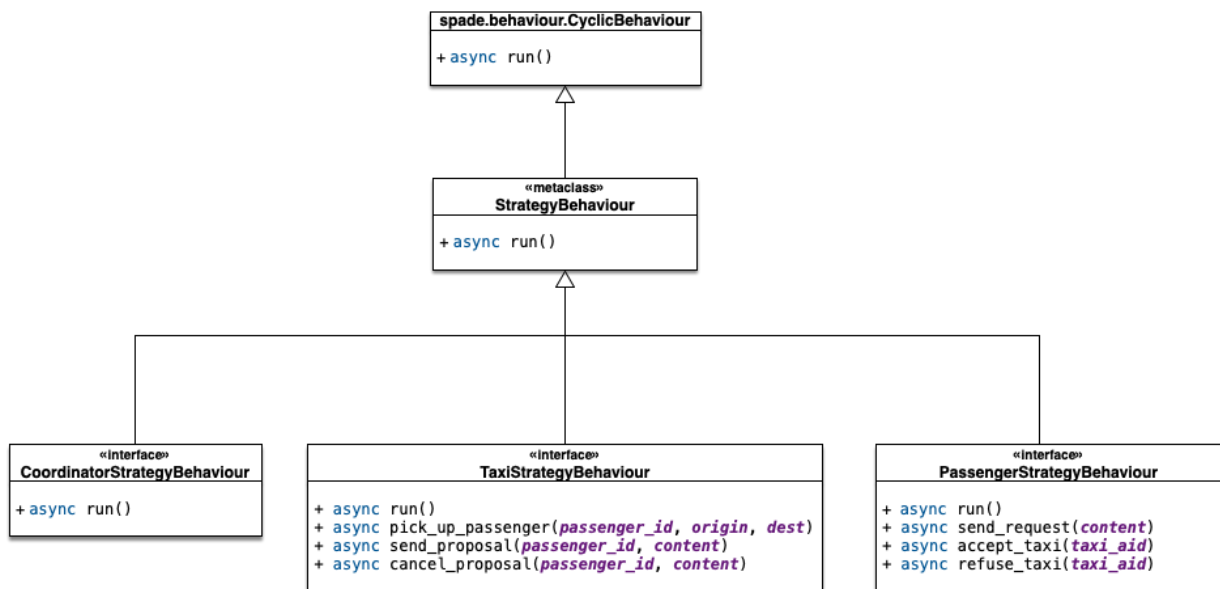Fig. 3: The Strategy Pattern UML.

Fig. 4: The StrategyBehaviour class and their inherited interfaces.

4.3.2 The Strategy Behaviour

The `StrategyBehaviour` is the metaclass from which interfaces are created for the strategies of each agent in the simulator. It inherits from a `spade.behaviour.CyclicBehaviour` class, so when implementing it, you will have to overload the `run` coroutine that will run cyclically (and endlessly), until the agent stops.

Helpers

The Strategy Behaviour provides also some helper functions that are useful in general for any kind of agent in the simulator.

> **Danger:** Don't store information in the Behaviour itself since it is a cyclic behaviour and is run by calling repeatedly the `run` coroutine, so the context of the function is not persistent. Use the agent variable that is accesible from any behaviour as *self.agent*. (i.e. you can do `self.agent.set("my_key", "my_value")` and `self.agent.get("my_key")`.

The `set` and `get` functions allow to store persistent information in the agent and to recover it at any moment. The store uses a *key-value* interface to store custom-defined data.

There is also a very useful helper function which is the **logger**. This is not a single function but a system of logs which can be used to generate debug information at different levels. There are four levels of logging which are now presented, in order of importance:

- **DEBUG** Used with `self.logger.debug("my debug message")`. These messages are only shown when the simulator is called with the `-v` option. This is usually superfluous information.
- **INFO** Used with `self.logger.info("my info message")`. These messages are always shown and are the regular information shown in logs.
- **WARNING** Used with `self.logger.warn("my warning message")`. These messages are always shown and are used to show warnings to the user.
- **ERROR** Used with `self.logger.error("my error message")`. These messages are always shown are are used to show errors to the user.

4.3.3 Developing the Coordinator Agent Strategy

In order to develop a new strategy for the Coordinator Agent, you need to create a class that inherits from `CoordinatorStrategyBehaviour`. Since this is a cyclic behaviour class that follows the *Strategy Pattern* and that inherits from the `StrategyBehaviour`, it has all the previously presented helper functions for communication and storing data inside the agent.

Following the *REQUEST* protocol, the Coordinator agent is supposed to receive every request for a taxi service from passengers and to carry out the action that your strategy determines (note that, in the default strategy `DelegateRequestTaxiBehaviour`, the coordinator delegates the decision to the taxis themselves by redirecting all requests to all taxis without any previous, additional reasoning). The code of the `DelegateRequestTaxiBehaviour` is presented below.

The place in the code where your coordinator strategy must be coded is the `run` coroutine. This function is executed in an infinite loop until the agent stops. In addition, you may also overload the `on_start` and the `on_end` coroutines, in order to execute code before the creation of the strategy or after its destruction, if needed.

Code

This is the code of the default coordinator strategy `DelegateRequestTaxiBehaviour`:

```python
from taxi_simulator.coordinator import CoordinatorStrategyBehaviour


class DelegateRequestTaxiBehaviour(CoordinatorStrategyBehaviour):

    async def run(self):
        msg = await self.receive(timeout=5)
        if msg:
            for taxi in self.get_taxi_agents():
                msg.to = str(taxi.jid)
                self.logger.debug("Coordinator sent request to taxi {}".format(taxi.
→name))
                await self.send(msg)
```

Helpers

The coordinator agent incorporates two helper functions that allow the agent to recover a list of all the taxi agents and passenger agents registered in the system. These functions are:

- `get_taxi_agents`

    Returns a list of the taxi agents.

- `get_passenger_agents`

    Returns a list of the passenger agents.

4.3.4 Developing the Taxi Agent Strategy

To develop a new strategy for the Taxi Agent, you need to create a class that inherits from `TaxiStrategyBehaviour`. Since this is a cyclic behaviour class that follows the *Strategy Pattern* and that inherits from the `StrategyBehaviour`, it has all the previously presented helper functions for communication and storing data inside the agent.

The taxi strategy is intended to receive requests from passengers, forwarded by the coordinator agent, and then to send proposals to these passengers in order to be selected by the corresponding passenger. If a taxi proposal is accepted, then the taxi begins the process of going to the passenger's current position, picking the passenger up, and taking the passenger to the requested destination.

> **Warning:** The process that implies a taxi movement is out of the scope of the strategy and should not be addressed by the strategy implementation. This pasenger-transfer process is automatically triggered when the strategy executes the helper coroutine `pick_up_passenger` (which is supposed to be the last action of a taxi strategy).

The place in the code where your coordinator strategy must be coded is the `run` coroutine. This function is executed in an infinite loop until the agent stops. In addition, you may also overload the `on_start` and the `on_end` coroutines, in order to execute code before the creation of the strategy or after its destruction, if needed.

Code

The default strategy of a taxi is to accept every passenger's requests if the taxi is not assigned to any other passenger or waiting a confirmation from any passenger. This is the code of the default taxi strategy

AcceptAlwaysStrategyBehaviour:

```python
from taxi_simulator.taxi import TaxiStrategyBehaviour


class AcceptAlwaysStrategyBehaviour(TaxiStrategyBehaviour):

    async def run(self):
        msg = await self.receive(timeout=5)
        if not msg:
            return
        self.logger.info("Taxi received message: {}".format(msg))
        content = json.loads(msg.body)
        performative = msg.get_metadata("performative")

        self.logger.debug("Taxi {} received request protocol from passenger {}.".
→format(self.agent.name,

→   content["passenger_id"]))
        if performative == REQUEST_PERFORMATIVE:
            if self.agent.status == TAXI_WAITING:
                await self.send_proposal(content["passenger_id"], {})
                self.agent.status = TAXI_WAITING_FOR_APPROVAL

        elif performative == ACCEPT_PERFORMATIVE:
            if self.agent.status == TAXI_WAITING_FOR_APPROVAL:
                self.logger.debug("Taxi {} got accept from {}".format(self.agent.name,
                                                                 content[
→"passenger_id"]))
                try:
                    self.agent.status = TAXI_MOVING_TO_PASSENGER
                    await self.pick_up_passenger(content["passenger_id"], content[
→"origin"], content["dest"])
                except PathRequestException:
                    self.logger.error("Taxi {} could not get a path to passenger {}.
→Cancelling..."
                                      .format(self.agent.name, content["passenger_id
→"]))
                    self.agent.status = TAXI_WAITING
                    await self.cancel_proposal(content["passenger_id"])
                except Exception as e:
                    self.logger.error("Unexpected error in taxi {}: {}".format(self.
→agent.name, e))
                    await self.cancel_proposal(content["passenger_id"])
                    self.agent.status = TAXI_WAITING
            else:
                await self.cancel_proposal(content["passenger_id"])

        elif performative == REFUSE_PERFORMATIVE:
            self.logger.debug("Taxi {} got refusal from {}".format(self.agent.name,
                                                             content["passenger_
→id"]))
            if self.agent.status == TAXI_WAITING_FOR_APPROVAL:
                self.agent.status = TAXI_WAITING
```

Helpers

There are some helper coroutines that are specific for the taxi strategy:

```
async def send_proposal(self, passenger_id, content=None)
async def cancel_proposal(self, passenger_id, content=None)
async def pick_up_passenger(self, passenger_id, origin, dest)
```

The definition and purpose of each of them is now introduced:

- `send_proposal`

    This helper function simplifies the composition and sending of a message containing a proposal to a passenger. It sends a `Message` to `passenger_id` using the **REQUEST_PROTOCOL** and a **PROPOSE_PERFORMATIVE**. It optionally accepts a *content* parameter where you can include any additional information you may want the passenger to analyze.

- `cancel_proposal`

    This helper function simplifies the composition and sending of a message to a passenger to cancel a proposal. It sends a `Message` to `passenger_id` using the **REQUEST_PROTOCOL** and a **CANCEL_PERFORMATIVE**. It optionally accepts a *content* parameter where you can include any additional information you may want the passenger to analyze.

- `pick_up_passenger`

    This helper function triggers the **TRAVEL_PROTOCOL** of a taxi, which is the protocol that is used to transport a passenger from her current position to her destination. This is a very important and particular function. Invoking this function is normally the last instruction of this strategy, since it means that the purpose of the strategy is accomplished (until the **TRAVEL_PROTOCOL** ends and the taxi is again free and able to receive new requests from some other passengers).

    The `pick_up_passenger` helper receives as parameters the id of the passenger and the coordinates of the passenger's current position (`origin`) and its destination (`dest`).

4.3.5 Developing the Passenger Agent Strategy

To develop a new strategy for the Passenger Agent, you need to create a class that inherits from `PassengerStrategyBehaviour`. Since this is a cyclic behaviour class that follows the *Strategy Pattern* and that inherits from the `StrategyBehaviour`, it has all the previously presented helper functions for communication and storing data inside the agent.

The passenger strategy is intended to ask the coordinator agent for a taxi service, then wait for taxi proposals and, after evaluating them, choosing a particular taxi proposal which will take the passenger to her destination.

The place in the code where your coordinator strategy must be coded is the `run` coroutine. This function is executed in an infinite loop until the agent stops. In addition, you may also overload the `on_start` and the `on_end` coroutines, in order to execute code before the creation of the strategy or after its destruction, if needed.

Code

The default strategy of a Passenger agent is a dummy strategy that simply accepts the first proposal it receives. This is the code of the default passenger strategy `AcceptFirstRequestTaxiBehaviour`:

```python
from taxi_simulator.passenger import PassengerStrategyBehaviour


class AcceptFirstRequestTaxiBehaviour(PassengerStrategyBehaviour):

    async def run(self):
        if self.agent.status == PASSENGER_WAITING:
```

```
                await self.send_request(content={})

        msg = await self.receive(timeout=5)

        if msg:
            performative = msg.get_metadata("performative")
            taxi_id = msg.sender
            if performative == PROPOSE_PERFORMATIVE:
                if self.agent.status == PASSENGER_WAITING:
                    self.logger.debug("Passenger {} received proposal from taxi {}".
→format(self.agent.name,

                                                                                    ␣
→     taxi_id))
                    await self.accept_taxi(taxi_id)
                    self.agent.status = PASSENGER_ASSIGNED
                else:
                    await self.refuse_taxi(taxi_id)

            elif performative == CANCEL_PERFORMATIVE:
                if self.agent.taxi_assigned == str(taxi_id):
                    self.logger.warning("Passenger {} received a CANCEL from Taxi {}.
→".format(self.agent.name, taxi_id))
                    self.agent.status = PASSENGER_WAITING
```

Helpers

There are some helper coroutines that are specific for the passenger strategy:

```
async def send_request(self, content=None)
async def accept_taxi(self, taxi_aid)
async def refuse_taxi(self, taxi_aid)
```

The definition and purpose of each of them is now introduced:

- send_request

    This helper is useful to make a new request without building the entire message (the function makes it for you). It creates a Message with a **REQUEST** performative and sends it to the coordinator agent. In addition, you can append a content to the request message to be used by the coordinator agent or the taxi agents (e.g. your origin coordinates or your destination coordinates).

- accept_taxi

    This is a helper function to send an acceptance message to a taxi_id. It sends a Message with an **ACCEPT** performative to the selected taxi.

- refuse_taxi

    This is a helper function to refuse a proposal from a taxi_id. It sends a Message with an **REFUSE** performative to the taxi whose proposal is being refused.

4.3.6 Other Helpers

Taxi Simulator also includes a helpers module which provides some general support methods that may be useful for any agent. These functions are now introduced:

- random_position

This helper function returns a random position in the map for being used if you need to create a new coordinate.

Example:

```
assert random_position() == [39.253, -0.341]
```

- are_close

  This helper function facilitates working with distances in maps. This helper function accepts two coordinates (coord1 and coord2) and an optional parameter to set the tolerance in meters. It returns True if both coordinates are closer than the tolerance in meters (10 meters by default). Otherwise it returns False.

  Example:

```
assert are_close([39.253, -0.341], [39.351, -0.333], 1000) == True
```

- distance_in_meters

  This helper function returns the distance in meters between two points.

  Example:

```
assert distance_in_meters([-0.37565, 39.44447], [-0.40392, 39.45293]) ==␣
↪3264.7134341427977
```

4.4 How to Implement New Strategies (Level 1) – Recommendations

At this point is time for you to implement your own strategies to optimize the problem of dispatching taxis to passengers. In this chapter we have shown you the tools to create these strategies. You have to create a file (in this example we are using my_strategy_file.py) and develop the strategies to be tested following the next template:

```python
from taxi_simulator.coordinator import CoordinatorStrategyBehaviour
from taxi_simulator.passenger import PassengerStrategyBehaviour
from taxi_simulator.taxi import TaxiStrategyBehaviour


################################################################
#                                                              #
#                   Coordinator Strategy                       #
#                                                              #
################################################################
class MyCoordinatorStrategy(CoordinatorStrategyBehaviour):
    async def run(self):
        # Your code here


################################################################
#                                                              #
#                       Taxi Strategy                          #
#                                                              #
################################################################
class MyTaxiStrategy(TaxiStrategyBehaviour):
    async def run(self):
        # Your code here


################################################################
#                                                              #
```

```
#                          Passenger Strategy                         #
#                                                                     #
#######################################################################
class MyPassengerStrategy(PassengerStrategyBehaviour):
    async def run(self):
        # Your code here
```

In this file, three strategies have been created for the three types of agent handled by the simulator. We have called these strategies `MyCoordinatorStrategy`, `MyTaxiStrategy` and `MyPassengerStrategy`.

To run the simulator with your new strategies the command line interface accepts three parameters with the name of the file (without extension) and the name of the class of each strategy.

```
$ taxi_simulator --host 127.0.0.1
                 --taxi my_strategy_file.MyTaxiStrategy
                 --passenger my_strategy_file.MyPassengerStrategy
                 --coordinator my_strategy_file.MyCoordinatorStrategy
```

> **Warning:** The file must be in the current working directory and it must be referenced *without* the extension (if the file is named `my_strategy_file.py` use `my_strategy_file` when calling the simulator.

Once run the simulator you can test your strategies using the graphical web interface or by inspecting the output of the logs in the command line.

API Documentation

Information on specific functions, classes, and methods.

5.1 taxi_simulator package

5.1.1 Submodules

5.1.2 taxi_simulator.cli module

Console script for taxi_simulator.

5.1.3 taxi_simulator.coordinator module

class taxi_simulator.coordinator.**CoordinatorAgent**(*agentjid*, *password*, *http_port*, *ip_address*)

    Bases: spade.agent.Agent

    Coordinator agent that manages the requests between taxis and passengers

    add_passenger(*agent*)
        Adds a new PassengerAgent to the store.

            Parameters agent (PassengerAgent) – the instance of the PassengerAgent to be added

    add_strategy(*strategy_class*)
        Injects the strategy by instantiating the strategy_class. Since the strategy_class inherits from spade.Behaviour.Behaviour, the new strategy is added as a behaviour to the agent.

            Parameters strategy_class (*class*) – the class to be instantiated.

    add_taxi(*agent*)
        Adds a new TaxiAgent to the store.

            Parameters agent (TaxiAgent) – the instance of the TaxiAgent to be added

async_create_agent(*cls*, *name*, *password*, *position*, *target*, *speed*)
    Coroutine to create an agent.

> **Parameters**
>
> * **cls** (`class`) – class of the agent (TaxiAgent or PassengerAgent)
> * **name** (`str`) – name of the agent
> * **password** (`str`) – password of the agent
> * **position** (`list`) – initial coordinates of the agent
> * **target** (`list, optional`) – destination coordinates of the agent
> * **speed** (`float, optional`) – speed of the vehicle

clean_controller(*request*)
    Web controller that resets the simulator to a clean state.

> **Returns** no template is returned since this is an AJAX controller, a dict with status=done
>
> **Return type** dict

clear_agents()
    Resets the set of taxis and passengers. Resets the simulation clock.

clear_stopped_agents()
    Removes from the taxi and passenger sets every agent that is stopped.

create_agent(*cls*, *name*, *password*, *position*, *target=None*, *speed=None*)
    Create an agent of type `cls` (TaxiAgent or PassengerAgent).

> **Parameters**
>
> * **cls** (`class`) – class of the agent (TaxiAgent or PassengerAgent)
> * **name** (`str`) – name of the agent
> * **password** (`str`) – password of the agent
> * **position** (`list`) – initial coordinates of the agent
> * **target** (`list, optional`) – destination coordinates of the agent
> * **speed** (`float, optional`) – speed of the vehicle

create_agents_batch(*cls*, *number: int*)
    Creates a batch of agents.

> **Parameters**
>
> * **cls** (`class`) – class of the agent to create
> * **number** (`int`) – size of the batch

download_stats_excel_controller(*request*)
    Web controller that returns an Excel file with the simulation results.

> **Returns** a Response of type "attachment" with the file content.
>
> **Return type** Response

download_stats_json_controller(*request*)
    Web controller that returns a JSON file with the simulation results.

> **Returns** a Response of type "attachment" with the file content.
>
> **Return type** Response

entities_controller(*request*)

    Web controller that returns a dict with the entities of the simulator and their statuses.

Example of the entities returned data:

```
{
    "passengers": [
        {
            "status": 24,
            "taxi": "taxi2@127.0.0.1",
            "dest": [ 39.463356, -0.376463 ],
            "waiting": 3.25,
            "position": [ 39.460568, -0.352529 ],
            "id": "michaelstewart"
        }
    ],
    "taxis": [
        {
            "status": 11,
            "passenger": "michaelstewart@127.0.0.1",
            "assignments": 1,
            "path": [
                    [ 39.478328, -0.406712 ],
                    [ 39.478317, -0.406814 ],
                    [ 39.460568, -0.352529 ]
                ],
            "dest": [ 39.460568, -0.352529 ],
            "position": [ 39.468131, -0.39685 ],
            "speed": 327.58,
            "id": "taxi2",
            "distance": "6754.60"
        }
    ],
    "stats": {
        "totaltime": "-1.00",
        "waiting": "3.25",
        "finished": False,
        "is_running": True
    },
    "tree": {
        "name": "Agents",
        "children": [
            {
                "count": "1",
                "name": "Taxis",
                "children": [ { "status": 11, "name": " taxi2", "icon": "fa-
→taxi" } ]
            },
            {
                "count": "1",
                "name": "Passengers",
                "children": [ { "status": 24, "name": " michaelstewart", "icon
→": "fa-user" } ]
            }
        ]
    },
    "authenticated": False
}
```

> **Returns** no template is returned since this is an AJAX controller, a dict with the list of taxis, the list of passengers, the tree view to be showed in the sidebar and the stats of the simulation.
>
> **Return type** dict

generate_controller(*request*)

generate_tree()
Generates the tree view in JSON format to be showed in the sidebar.

> **Returns** a dict with all the agents in the simulator, with their name, status and icon.
>
> **Return type** dict

get_passenger_stats()
Creates a dataframe with the simulation stats of the passengers The dataframe includes for each passenger its name, waiting time, total time and status.

> **Returns** the dataframe with the passengers stats.
>
> **Return type** `pandas.DataFrame`

get_simulation_time()
Returns the elapsed simulation time to the current time. If the simulation is not started it returns 0.

> **Returns** the whole simulation time.
>
> **Return type** float

get_stats()
Generates the stats of the simulation in JSON format.

Examples:

```
{
    "totaltime": "12.25",
    "waiting": "3.25",
    "finished": False,
    "is_running": True
}
```

> **Returns** a dict with the total time, waiting time, is_running and finished values
>
> **Return type** dict

get_stats_dataframes()
Collects simulation stats and returns 3 dataframes with the information: A general dataframe with the average information, a dataframe with the taxi's information and a dataframe with the passenger's information. :returns: avg df, taxi df and passenger df :rtype: pandas.Dataframe, pandas.Dataframe, pandas.Dataframe

get_taxi_stats()
Creates a dataframe with the simulation stats of the taxis The dataframe includes for each taxi its name, assignments, traveled distance and status.

> **Returns** the dataframe with the taxis stats.
>
> **Return type** `pandas.DataFrame`

index_controller(*request*)
Web controller that returns the index page of the simulator.

> **Returns** the name of the template, the data to be pre-processed in the template
>
> **Return type** dict

`is_simulation_finished``()`

Checks whether the simulation has finished or not. A simulation is finished if all passengers are at their destinations. If there is no passengers the simulation is not finished.

> **Returns** whether the simulation has finished or not.

> **Return type** bool

`passenger_agents`

Gets the dict of registered passengers

> **Returns** a dict of `PassengerAgent` with the name in the key

> **Return type** dict

`request_path`(*origin*, *destination*)

Requests a path to the RouteAgent.

> **Parameters**
>
> - **`origin`** (`list`) – the origin coordinates (lon, lat)
>
> - **`destination`** (`list`) – the target coordinates (lon, lat)

> **Returns** the path as a list of points, the distance of the path, the estimated duration of the path

> **Return type** list, float, float

`run_controller`(*request*)

Web controller that starts the simulator.

> **Returns** no template is returned since this is an AJAX controller, an empty data dict is returned

> **Return type** dict

`run_simulation`()

Starts the simulation

`set_strategies`(*coordinator_strategy*, *taxi_strategy*, *passenger_strategy*)

Gets the strategy strings and loads their classes. This strategies are prepared to be injected into any new taxi or passenger agent.

> **Parameters**
>
> - **`coordinator_strategy`** (`str`) – the path to the coordinator strategy
>
> - **`taxi_strategy`** (`str`) – the path to the taxi strategy
>
> - **`passenger_strategy`** (`str`) – the path to the passenger strategy

`setup`()

Setup agent before startup. This method may be overloaded.

`stop_agents`()

Stops the simulator and all the agents

`stop_agents_controller`(*request*)

Web controller that stops all the passenger and taxi agents.

> **Returns** no template is returned since this is an AJAX controller, a dict with status=done

> **Return type** dict

`taxi_agents`

Gets the dict of registered taxis

> **Returns** a dict of `TaxiAgent` with the name in the key

Return type dict

class taxi_simulator.coordinator.**CoordinatorStrategyBehaviour**
Bases: *taxi_simulator.utils.StrategyBehaviour*

Class from which to inherit to create a coordinator strategy. You must overload the _process() method

Helper functions:

- *get_taxi_agents()*

- *get_passenger_agents()*

get_passenger_agents()
Gets the list of registered passengers

Returns a list of PassengerAgent

Return type list

get_taxi_agents()
Gets the list of registered taxis

Returns a list of TaxiAgent

Return type list

on_start()
Coroutine called before the behaviour is started.

run()
Body of the behaviour. To be implemented by user.

5.1.4 taxi_simulator.helpers module

Helpers module

These functions are useful for the develop of new strategies.

exception taxi_simulator.helpers.**AlreadyInDestination**
Bases: Exception

This exception is raised when an agent wants to move to a destination where it is already there.

exception taxi_simulator.helpers.**PathRequestException**
Bases: Exception

This exception is raised when a path could not be computed.

taxi_simulator.helpers.**are_close**(*coord1*, *coord2*, *tolerance=10*)
Checks wheter two points are close or not. The tolerance is expressed in meters.

Parameters

- **coord1** (*list*) – a coordinate (longitude, latitude)

- **coord2** (*list*) – another coordinate (longitude, latitude)

- **tolerance** (*int*) – tolerance in meters

Returns whether the two coordinates are closer than tolerance or not

Return type bool

taxi_simulator.helpers.**distance_in_meters**(*coord1*, *coord2*)
Returns the distance between two coordinates in meters.

> **Parameters**
>> * **coord1** (*list*) – a coordinate (longitude, latitude)
>> * **coord2** – another coordinate (longitude, latitude)
>
> **Returns** distance meters between the two coordinates
>
> **Return type** float

`taxi_simulator.helpers.` **`kmh_to_ms`** (*speed_in_kmh*)
> Convert kilometers/hour to meters/second.
>
>> **Parameters** **`speed_in_kmh`** (*float*) – speed in kilometers/hour
>>
>> **Returns** the speed in meters/second
>>
>> **Return type** float

`taxi_simulator.helpers.` **`random_position`** ()
> Returns a random position inside the map.
>
>> **Returns** a point (longitude and latitude)
>>
>> **Return type** list

5.1.5 taxi_simulator.passenger module

class `taxi_simulator.passenger.` **`PassengerAgent`** (*agentjid*, *password*, *loop=None*)
> Bases: `spade.agent.Agent`

> **`add_strategy`** (*strategy_class*)
>> Sets the strategy for the passenger agent.
>>
>>> **Parameters** **`strategy_class`** (`PassengerStrategyBehaviour`) – The class to be used. Must inherit from `PassengerStrategyBehaviour`

> **`get_pickup_time`** ()
>> Returns the time that the passenger was waiting to be picked up since it has been assigned to a taxi.
>>
>>> **Returns** The time that the passenger was waiting to a taxi since it has been assigned.
>>>
>>> **Return type** float

> **`get_position`** ()
>> Returns the current position of the passenger.
>>
>>> **Returns** the coordinates of the current position of the passenger (lon, lat)
>>>
>>> **Return type** list

> **`get_waiting_time`** ()
>> Returns the time that the agent was waiting for a taxi, from its creation until it gets into a taxi.
>>
>>> **Returns** The time the passenger was waiting.
>>>
>>> **Return type** float

> **`is_in_destination`** ()
>> Checks if the passenger has arrived to its destination.
>>
>>> **Returns** whether the passenger is at its destination or not
>>>
>>> **Return type** bool

request_path(*origin*, *destination*)
  Requests a path between two points (origin and destination) using the RouteAgent service.

  Parameters

  - **origin** (`list`) – the coordinates of the origin of the requested path

  - **destination** (`list`) – the coordinates of the end of the requested path

  Returns A list of points that represent the path from origin to destination, the distance and the estimated duration

  Return type list, float, float

set_coordinator(*coordinator_id*)
  Sets the coordinator JID address :param coordinator_id: the coordinator jid :type coordinator_id: str

set_id(*agent_id*)
  Sets the agent identifier :param agent_id: The new Agent Id :type agent_id: str

set_position(*coords=None*)
  Sets the position of the passenger. If no position is provided it is located in a random position.

  Parameters coords (`list`) – a list coordinates (longitude and latitude)

set_route_agent(*route_id*)
  Sets the route agent JID address :param route_id: the route agent jid :type route_id: str

set_target_position(*coords=None*)
  Sets the target position of the passenger (i.e. its destination). If no position is provided the destination is setted to a random position.

  Parameters coords (`list`) – a list coordinates (longitude and latitude)

setup()
  Setup agent before startup. This method may be overloaded.

to_json()
  Serializes the main information of a passenger agent to a JSON format. It includes the id of the agent, its current position, the destination coordinates of the agent, the current status, the taxi that it has assigned (if any) and its waiting time.

  Returns

  a JSON doc with the main information of the passenger.

  Example:

```
{
    "id": "cphillips",
    "position": [ 39.461327, -0.361839 ],
    "dest": [ 39.460599, -0.335041 ],
    "status": 24,
    "taxi": "ghiggins@127.0.0.1",
    "waiting": 13.45
}
```

  Return type dict

total_time()
  Returns the time since the passenger was activated until it reached its destination.

  Returns the total time of the passenger's simulation.

  Return type float

class taxi_simulator.passenger.**PassengerStrategyBehaviour**
  Bases: *taxi_simulator.utils.StrategyBehaviour*

  Class from which to inherit to create a taxi strategy. You must overload the run coroutine

  Helper functions:

  - send_request
  - accept_taxi
  - refuse_taxi

  accept_taxi(*taxi_id*)
    Sends a spade.message.Message to a taxi to accept a travel proposal. It uses the RE-
    QUEST_PROTOCOL and the ACCEPT_PERFORMATIVE.

        Parameters taxi_id(*str*) – The Agent JID of the taxi

  on_start()
    Initializes the logger and timers. Call to parent method if overloaded.

  refuse_taxi(*taxi_id*)
    Sends an spade.message.Message to a taxi to refuse a travel proposal. It uses the RE-
    QUEST_PROTOCOL and the REFUSE_PERFORMATIVE.

        Parameters taxi_id(*str*) – The Agent JID of the taxi

  run()
    Body of the behaviour. To be implemented by user.

  send_request(*content=None*)
    Sends an spade.message.Message to the coordinator to request a taxi. It uses the RE-
    QUEST_PROTOCOL and the REQUEST_PERFORMATIVE. If no content is set a default content with
    the passenger_id, origin and target coordinates is used.

        Parameters content(*dict*) – Optional content dictionary

class taxi_simulator.passenger.**TravelBehaviour**
  Bases: spade.behaviour.CyclicBehaviour

  This is the internal behaviour that manages the movement of the passenger. It is triggered when the taxi informs
  the passenger that it is going to the passenger's position until the passenger is droppped in its destination.

  on_start()
    Coroutine called before the behaviour is started.

  run()
    Body of the behaviour. To be implemented by user.

5.1.6 taxi_simulator.protocol module

protocol and performative constants

5.1.7 taxi_simulator.route module

class taxi_simulator.route.**RouteAgent**(*agentjid*, *password*)
  Bases: spade.agent.Agent

  The RouteAgent receives request for paths, queries an OSRM server and returns the information. It also caches
  the queries to avoid overloading the OSRM server.

class RequestRouteBehaviour
    Bases: `spade.behaviour.CyclicBehaviour`

    This cyclic behaviour listens for route requests from other agents. When a message is received it answers with the path.

    on_end()
        Coroutine called after the behaviour is done or killed.

    on_start()
        Coroutine called before the behaviour is started.

    run()
        Body of the behaviour. To be implemented by user.

get_route(*origin*, *destination*)
    Checks the cache for a path, if not found then it queries the OSRM server.

        Parameters

            • **origin** (`list`) – origin coordinate (longitude, latitude)

            • **destination** (`list`) – target coordinate (longitude, latitude)

        Returns a dict with three keys: path, distance and duration

        Return type dict

load_cache()
    Loads the cache from file.

persist_cache()
    Persists the cache to a JSON file.

static request_route_to_server(*origin*, *destination*)
    Queries the OSRM for a path.

        Parameters

            • **origin** (`list`) – origin coordinate (longitude, latitude)

            • **destination** (`list`) – target coordinate (longitude, latitude)

        Returns list, float, float = the path, the distance of the path and the estimated duration

setup()
    Setup agent before startup. This method may be overloaded.

5.1.8 taxi_simulator.scenario module

class `taxi_simulator.scenario.`**Scenario**(*filename*)
    Bases: `object`

    A scenario object reads a file with a JSON representation of a scenario and is used to create the participant agents.

    load(*coordinator: taxi_simulator.coordinator.CoordinatorAgent*)

5.1.9 taxi_simulator.simulator module

class `taxi_simulator.simulator.`**SimulationConfig**
    Bases: `object`

Dataclass to store the *Simulator* config

class taxi_simulator.simulator.**Simulator**(*config*)

Bases: object

The Simulator. It manages all the simulation processes. Tasks done by the simulator at initialization:

1. Create the XMPP server

2. Run the SPADE backend

3. Run the coordinator and route agents.

4. Create agents passed as parameters (if any).

5. Create agents defined in scenario (if any).

After these tasks are done in the Simulator constructor, the simulation is started when the run method is called.

collect_stats()

Collects stats from all participant agents and from the simulation and stores it in three dataframes.

get_stats()

Returns the dataframes collected by *collect_stats()*

> **Returns** average df, passengers df and taxi df
>
> **Return type** pandas.DataFrame, pandas.DataFrame, pandas.DataFrame

is_simulation_finished()

Checks if the simulation is finished. A simulation is finished if the max simulation time has been reached or when the coordinator says it.

> **Returns** whether the simulation is finished or not.
>
> **Return type** bool

print_stats()

Prints the dataframes collected by collect_stats.

run()

Starts the simulation (tells the coordinator agent to start the simulation).

stop()

Finishes the simulation and prints simulation stats. Tasks done when a simulation is stopped:

1. Stop participant agents.

2. Print stats.

3. Stop Route agent.

4. Stop Coordinator agent.

time_is_out()

Checks if the max simulation time has been reached.

> **Returns** whether the max simulation time has been reached or not.
>
> **Return type** bool

write_excel(*filename*)

Writes the collected data by collect_stats in an excel file.

> **Parameters** **filename** (*str*) – name of the excel file.

> **write_file**(*filename*, *fileformat='json'*)
>   Writes the dataframes collected by `collect_stats` in JSON or Excel format.
>
> > **Parameters**
> >
> > - **filename** (*str*) – name of the output file to be written.
> >
> > - **fileformat** (*str*) – format of the output file. Choices: json or excel
>
> **write_json**(*filename*)
>   Writes the collected data by `collect_stats` in a json file.
>
> > **Parameters** **filename** (*str*) – name of the json file.

5.1.10 taxi_simulator.strategies module

class `taxi_simulator.strategies.`**AcceptAlwaysStrategyBehaviour**
    Bases: *taxi_simulator.taxi.TaxiStrategyBehaviour*

> The default strategy for the Taxi agent. By default it accepts every request it receives if available.
>
> **run**()
>   Body of the behaviour. To be implemented by user.

class `taxi_simulator.strategies.`**AcceptFirstRequestTaxiBehaviour**
    Bases: *taxi_simulator.passenger.PassengerStrategyBehaviour*

> The default strategy for the Passenger agent. By default it accepts the first proposal it receives.
>
> **run**()
>   Body of the behaviour. To be implemented by user.

class `taxi_simulator.strategies.`**DelegateRequestTaxiBehaviour**
    Bases: *taxi_simulator.coordinator.CoordinatorStrategyBehaviour*

> The default strategy for the Coordinator agent. By default it delegates all requests to all taxis.
>
> **run**()
>   Body of the behaviour. To be implemented by user.

5.1.11 taxi_simulator.strategies_fsm module

class `taxi_simulator.strategies_fsm.`**FSMTaxiStrategyBehaviour**
    Bases: spade.behaviour.FSMBehaviour

> **setup**()

class `taxi_simulator.strategies_fsm.`**TaxiMovingState**
    Bases: *taxi_simulator.taxi.TaxiStrategyBehaviour*, spade.behaviour.State

> **on_start**()
>   Coroutine called before the behaviour is started.
>
> **run**()
>   Body of the behaviour. To be implemented by user.

class `taxi_simulator.strategies_fsm.`**TaxiWaitingForApprovalState**
    Bases: *taxi_simulator.taxi.TaxiStrategyBehaviour*, spade.behaviour.State

> **on_start**()
>   Coroutine called before the behaviour is started.

> **run**()
> > Body of the behaviour. To be implemented by user.

class taxi_simulator.strategies_fsm.**TaxiWaitingState**
> Bases: *taxi_simulator.taxi.TaxiStrategyBehaviour*, spade.behaviour.State

> **on_start**()
> > Coroutine called before the behaviour is started.

> **run**()
> > Body of the behaviour. To be implemented by user.

taxi_simulator.strategies_fsm.**passenger_in_taxi_callback**(*old*, *new*)

5.1.12 taxi_simulator.taxi module

class taxi_simulator.taxi.**TaxiAgent**(*agentjid*, *password*, *loop=None*)
> Bases: spade.agent.Agent

> **class MovingBehaviour**(*period*, *start_at=None*)
> > Bases: spade.behaviour.PeriodicBehaviour

> > This is the internal behaviour that manages the movement of the taxi. It is triggered when the taxi has a new destination and the periodic tick is recomputed at every step to show a fine animation. This moving behaviour includes to update the taxi coordinates as it moves along the path at the specified speed.

> > **run**()
> > > Body of the behaviour. To be implemented by user.

> **add_strategy**(*strategy_class*)
> > Sets the strategy for the taxi agent.

> > > **Parameters strategy_class** (TaxiStrategyBehaviour) – The class to be used. Must inherit from TaxiStrategyBehaviour

> **arrived_to_destination**()
> > Informs that the taxi has arrived to its destination. It recomputes the new destination and path if picking up a passenger or drops it and goes to WAITING status again.

> **cancel_passenger**(*data=None*)
> > Sends a message to the current assigned passenger to cancel the assignment.

> > > **Parameters data** (*dict, optional*) – Complementary info about the cancellation

> **drop_passenger**()
> > Drops the passenger that the taxi is carring in the current location.

> **get_position**()
> > Returns the current position of the passenger.

> > > **Returns** the coordinates of the current position of the passenger (lon, lat)

> > > **Return type** list

> **inform_passenger**(*status*, *data=None*)
> > Sends a message to the current assigned passenger to inform her about a new status.

> > > **Parameters**
> > > - **status** (*int*) – The new status code
> > > - **data** (*dict, optional*) – complementary info about the status

> **is_free**()

is_in_destination()
>   Checks if the taxi has arrived to its destination.

>>   **Returns** whether the taxi is at its destination or not

>>   **Return type** bool

is_passenger_in_taxi()

move_to(*dest*)
>   Moves the taxi to a new destination.

>>   **Parameters dest** (*list*) – the coordinates of the new destination (in lon, lat format)

>>   **Raises** AlreadyInDestination – if the taxi is already in the destination coordinates.

request_path(*origin*, *destination*)
>   Requests a path between two points (origin and destination) using the RouteAgent service.

>>   **Parameters**

>>>   • **origin** (*list*) – the coordinates of the origin of the requested path

>>>   • **destination** (*list*) – the coordinates of the end of the requested path

>>   **Returns** A list of points that represent the path from origin to destination, the distance and the estimated duration

>>   **Return type** list, float, float

Examples

```
>>> path, distance, duration = await self.request_path(origin=[0,0],
↪destination=[1,1])
>>> print(path)
[[0,0], [0,1], [1,1]]
>>> print(distance)
2.0
>>> print(duration)
3.24
```

send(*msg*)

set_coordinator(*coordinator_id*)
>   Sets the coordinator JID address :param coordinator_id: the coordinator jid :type coordinator_id: str

set_id(*agent_id*)
>   Sets the agent identifier

>>   **Parameters agent_id** (*str*) – The new Agent Id

set_position(*coords=None*)
>   Sets the position of the taxi. If no position is provided it is located in a random position.

>>   **Parameters coords** (*list*) – a list coordinates (longitude and latitude)

set_route_agent(*route_id*)
>   Sets the route agent JID address :param route_id: the route agent jid :type route_id: str

set_speed(*speed_in_kmh*)
>   Sets the speed of the taxi.

>>   **Parameters speed_in_kmh** (*float*) – the speed of the taxi in km per hour

step()
> Advances one step in the simulation

to_json()
> Serializes the main information of a taxi agent to a JSON format. It includes the id of the agent, its current position, the destination coordinates of the agent, the current status, the speed of the taxi (in km/h), the path it is following (if any), the passenger that it has assigned (if any), the number of assignments if has done and the distance that the taxi has traveled.
>
> > **Returns**
> >
> > > a JSON doc with the main information of the taxi.
> >
> > Example:
> >
> > ```
> > {
> >     "id": "cphillips",
> >     "position": [ 39.461327, -0.361839 ],
> >     "dest": [ 39.460599, -0.335041 ],
> >     "status": 24,
> >     "speed": 1000,
> >     "path": [[0,0], [0,1], [1,0], [1,1], ...],
> >     "passenger": "ghiggins@127.0.0.1",
> >     "assignments": 2,
> >     "distance": 3481.34
> > }
> > ```
> >
> > **Return type** dict

watch_value(*key*, *callback*)
> Registers an observer callback to be run when a value is changed
>
> > **Parameters**
> >
> > - **key** (*str*) – the name of the value
> > - **callback** (*function*) – a function to be called when the value changes. It receives two arguments: the old and the new value.

class taxi_simulator.taxi.**TaxiStrategyBehaviour**
> Bases: *taxi_simulator.utils.StrategyBehaviour*

Class from which to inherit to create a taxi strategy. You must overload the `` `run `` coroutine

Helper functions:

- pick_up_passenger
- send_proposal
- cancel_proposal

cancel_proposal(*passenger_id*, *content=None*)
> Send a spade.message.Message to cancel a proposal. If the content is empty the proposal is sent without content.
>
> > **Parameters**
> >
> > - **passenger_id** (*str*) – the id of the passenger
> > - **content** (*dict, optional*) – the optional content of the message

on_start()
> Coroutine called before the behaviour is started.

---

pick_up_passenger(*passenger_id*, *origin*, *dest*)

> Starts a TRAVEL_PROTOCOL to pick up a passenger and get him to his destination. It automatically launches all the travelling process until the passenger is delivered. This travelling process includes to update the taxi coordinates as it moves along the path at the specified speed.
>
> > **Parameters**
> >
> > - **passenger_id** (`str`) – the id of the passenger
> >
> > - **origin** (`list`) – the coordinates of the current location of the passenger
> >
> > - **dest** (`list`) – the coordinates of the target destination of the passenger

run()

> Body of the behaviour. To be implemented by user.

send_proposal(*passenger_id*, *content=None*)

> Send a `spade.message.Message` with a proposal to a passenger to pick up him. If the content is empty the proposal is sent without content.
>
> > **Parameters**
> >
> > - **passenger_id** (`str`) – the id of the passenger
> >
> > - **content** (`dict, optional`) – the optional content of the message

5.1.13 taxi_simulator.utils module

class `taxi_simulator.utils.`**RequestRouteBehaviour**(*msg: spade.message.Message*, *origin: list*, *destination: list*, *route_agent: str*)

> Bases: `spade.behaviour.OneShotBehaviour`
>
> A one-shot behaviour that is executed to request for a new route to the route agent.
>
> **run**()
>
> > Body of the behaviour. To be implemented by user.

class `taxi_simulator.utils.`**StrategyBehaviour**

> Bases: `spade.behaviour.CyclicBehaviour`
>
> The behaviour that all parent strategies must inherit from. It complies with the Strategy Pattern.

`taxi_simulator.utils.`**avg**(*array*)

> Makes the average of an array without Nones. :param array: a list of floats and Nones :type array: list
>
> > **Returns** the average of the list without the Nones.
> >
> > **Return type** float

`taxi_simulator.utils.`**chunk_path**(*path*, *speed_in_kmh*)

> Splits the path into smaller chunks taking into account the speed.
>
> > **Parameters**
> >
> > - **path** (`list`) – the original path. A list of points (lon, lat)
> >
> > - **speed_in_kmh** (`float`) – the speed in km per hour at which the path is being traveled.
> >
> > **Returns** a new path equivalent (to the first one), that has at least the same number of points.
> >
> > **Return type** list

`taxi_simulator.utils.`**load_class**(*class_path*)

> Tricky method that imports a class form a string.

> **Parameters class_path** (*str*) – the path where the class to be imported is.
>
> **Returns** the class imported and ready to be instantiated.
>
> **Return type** class

taxi_simulator.utils.**request_path**(*agent*, *origin*, *destination*, *route_id*)

> Sends a message to the RouteAgent to request a path
>
> > **Parameters**
> >
> > - **agent** – the agent who is requesting the path
> > - **origin** (*list*) – a list with the origin coordinates [longitude, latitude]
> > - **destination** (*list*) – a list with the target coordinates [longitude, latitude]
> >
> > **Returns**
> >
> > > **a list of points (longitude and latitude) representing the path,** the distance of the path in meters, a estimation of the duration of the path
> >
> > **Return type** list, float, float

Examples

```
>>> path, distance, duration = request_path(an_agent, origin=[0,0],
→destination=[1,1])
>>> print(path)
[[0,0], [0,1], [1,1]]
>>> print(distance)
2.0
>>> print(duration)
3.24
```

taxi_simulator.utils.**status_to_str**(*status_code*)

> Translates an int status code to a string that represents the status
>
> > **Parameters status_code** (*int*) – the code of the status
> >
> > **Returns** the string that represents the status
> >
> > **Return type** str

taxi_simulator.utils.**unused_port**(*hostname*)

> Return a port that is unused on the current host.

5.1.14 Module contents

Top-level package for Taxi Simulator.

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/javipalanca/taxi_simulator/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

Taxi Simulator could always use more documentation, whether as part of the official Taxi Simulator docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/javipalanca/taxi_simulator/issues.

If you are proposing a feature:

- Explain in detail how it would work.

- Keep the scope as narrow as possible, to make it easier to implement.

- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *taxi_simulator* for local development.

1. Fork the *taxi_simulator* repo on GitHub.

2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/taxi_simulator.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv taxi_simulator
   $ cd taxi_simulator/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 taxi_simulator tests
   $ python setup.py test or py.test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.

2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.

3. The pull request should work for Python 3.6. Check https://travis-ci.org/javipalanca/taxi_simulator/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ py.test tests.test_taxi_simulator
```

Credits

7.1 Development Lead

- Javi Palanca <jpalanca@gmail.com>

7.2 Contributors

None yet. Why not be the first?

History

8.1 0.4.1 (2019-01-07)

- Fixed bug when checking if the simulation is finished.

8.2 0.4.0 (2018-10-25)

- Improved the concurrent creation of agents.
- Added stop and clear buttons to the interface.
- Added download button for getting results in excel and json formats.
- Documentation updated.

8.3 0.3.0 (2018-10-01)

- Migrated to SPADE 3.
- Documentation highly improved.
- Helper functions added and refined.
- Javascript framework included: VueJS
- Routes centralized with a Route agent.
- UI improved.

8.4 0.2 (2017-11-15)

- Added scenario loading feature.

8.5 0.1.3 (2017-11-15)

- Fixed minor bugs.

8.6 0.1.1 (2017-11-14)

- Added documentation.

8.7 0.1.0 (2017-11-03)

- First release on PyPI.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[GangOfFour95]   5. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Elements of Reusable Object
                Oriented Software. Addison-Wesley, 1995.

Python Module Index

t

Index