Document downloaded from:

http://hdl.handle.net/10251/176072

This paper must be cited as:

Palanca Cámara, J.; Terrasa Barrena, AM.; Rodriguez, S.; Carrascosa Casamayor, C.; Julian Inglada, VJ. (2021). An agent-based simulation framework for the study of urban delivery. Neurocomputing. 423:679-688. https://doi.org/10.1016/j.neucom.2020.03.117

The final publication is available at

https://doi.org/10.1016/j.neucom.2020.03.117

Copyright Elsevier

Additional Information

An agent-based simulation framework for the study of urban delivery

J. Palanca[a], A. Terrasa[a], S. Rodriguez[b], C. Carrascosa[a], V. Julian[a]

[a]*Institut Valencià d'Investigació en Intel·ligència Artificial (VRAIN)*
Universitat Politècnica de València
{jpalanca,aterrasa,carrasco,vinglada}@dsic.upv.es
[b]*BISITE, University of Salamanca, srg@usal.es*

Abstract

In recent years, cities and especially urban mobility have undergone remarkable changes. Significant advances in technology have been translated into new mobility services for both goods and people. One evident change has been the transformation of traditional vehicle fleets into more open fleets, in the sense that their members can proactively decide whether or not they are part of a certain fleet and whether or not they perform certain services. Fleets of this type make the decision-making process to be highly distributed, and rule out some of the typically centralized decisions. The management and control of this type of open fleets is severely more complex and, for this reason, the availability of simulation tools that allow for their analysis can be very useful. In accordance with this, the main contribution of this work is the development of an agent-based simulation tool specifically designed for the simulation of new urban mobility models. In this way, the tool can simulate any type of fleet in different urban scenarios, including a solution of the Last Mile Delivery problem, which is also included as a proof of concept in this paper.

Keywords: multi-agent systems; coordination; smart cities

1. Introduction

Over the last few years, the increase of transportation and mobility in the cities has become one of the main challenges facing us as a society. Urban mobility is currently one of the causes of air pollution, traffic jams, problems in

logistics and energy waste in current cities. Innovative solutions for communication networks, information processing and transport are currently being developed to meet this challenge. These developments could ensure the most efficient use of resources and provide flexible mobility solutions for citizens and businesses [1].

Current technology for public transport in urban environments has improved the quality of services offered to citizens. Existing technology provide public transport managers with real-time information of the public transport system. This allows service providers to manage their fleets in a more effective way [2]. In addition, these systems can provide up-to-date information to passengers or users [3] anywhere (vehicles, stops, at home, etc.) and form the basis of a complete Smart Transport System [4, 5].

Most urban mobility services in today's cities have been transformed into a new concept called "open fleet" [1]. An open fleet differs from traditional fleets in the sense that individuals have complete autonomy and there is not a centralized entity that governs the fleet. In an open fleet, vehicles may interact with their environment in a Smart city, and join or leave the fleet at any time. In any case, similarly to traditional fleets, an open fleet requires a global regulatory entity that manages and coordinates the use of a limited set of resources in order to provide a specific transportation service. The efficiency of an open fleet depends on the use of appropriated coordination and regulation mechanisms that deal with the problem of balancing global and individual objectives. Regarding the coordination problem of urban fleets, this has traditionally been studied for more closed fleets in different areas, and its impact, especially in the field of emergency services [6] or, in recent years, to the coordination of fleets for vehicle sharing [7].

According to this, urban mobility can be seen as a set of several autonomous entities that will act interchangeably as offering or demanding services. In this infrastructure, the entities will not be considered as just black-boxes, but will also get interconnected with the global goal to improve the coordination of the offered services. This view fits perfectly with the definition of a multi-agent system (MAS). With this idea in mind, we propose the use of agent-based simulation tools for the study and analysis of the more appropriated models, architectures or strategies for the management of new urban mobility models or strategies. Agent-based simulation (ABS) offers a way to model social systems composed of individuals that interact with each other, learning from their experiences and adapting their behaviors to achieve goals in the environment to which they have been destined, both

individually and collectively [8].

Taking these ideas into account, the main contribution of this work is the development of an agent-based toolkit for the simulation of new models or strategies in the urban mobility area and, specifically, for the problem of deliveries in the urban environment, including the management of new open fleets, as commented above. The simulation tool allows researchers or policy makers to analyze new coordination and regulation strategies related with urban mobility in order to obtain an efficient solution with regard to some globally desirable parameters. This paper is an extension of a previous work published in [9].

The rest of the paper is structured as follows. Section 2 analyzes some previous work related with the topic. Section 3 presents the proposed simulation tool for simulating fleet scenarios; Section 4 illustrates a specific use of the tool for the management of the last mile delivery problem. Finally, Section 5 presents the conclusions of this paper.

2. Related Work

In the literature, there are several different approaches which try to simulate, or propose strategies for, the urban delivery problem. Although most of the proposals focus on passenger mobility in the urban environment [10, 11], there are also solutions centered on the movement of packages. In [12], a review on transport, logistics, and mobility requirements in the Smart Cities environments can be found.

Multi-Agent System has been one of the main technologies being used for the development of approaches which are aimed at testing possible solutions or strategies to the urban delivery problem. From a general perspective, the best known simulation tools are SUMO [13] and MatSim [14]. SUMO can be considered a traffic flow simulation platform, including vehicles, any public transportation, and also pedestrians. It includes different tools and add-ons which offer functionalities such as locating routes or V2X communications. It provides multiple application programming interfaces to control the simulation in many ways. On the other hand, MatSim is an agent-based simulation framework for implementing large-scale agent-based transport simulations. MatSim allows the design of agents formed by a set of activities that represent different transport demands. Agent's decisions or strategies about how to travel are scheduled before the simulation. These simulation tools are typically oriented to general-purpose applications and their specific use to

design and evaluate strategies for the urban delivery problem can be highly complex.

In this sense, there are more specific proposals. An interesting approach is the work proposed in [15], where a MAS simulation model is introduced. This model is capable of optimizing the distribution phase of small and medium parcels. This proposal is based on a combination of different public transport systems and bikes for the delivery of parcels. The work presented in [16] uses a taxi fleet in a city in order to apply a *crowdsourcing* solution for the last mile delivery problem. The proposed model is a closed approach where the taxi drivers that are willing to deliver parcels must be first registered in the system.

Another related approach is the work proposed in [17], which presents another crowdsourcing approach used for library deliveries. The idea is that citizens deliver parcels to each other along their own ways. Results of the experiments showed a reduction of the distance needed for each delivery by using the crowdsourced approach. Finally, regarding urban delivery solutions, [18] proposes a MAS model for evaluating city logistics measures, which tries to measure logistics efficiency in a city with congested urban traffic conditions.

The use of delivery lockers (or similar solutions) is another, recent way which enables carriers to reduce the number of trucks required to make deliveries. In [19], an agent-based simulation model is presented which aims to reduce the number of re-deliveries through the use of delivery lockers which will be directly used in the third attempts of deliveries. The simulation confirms that the use of delivery lockers reduces the distance (kilometers) traveled by trucks, and also shows how a multi-agent simulation is an appropriate tool for modeling urban delivery transport scenarios. Moreover, the work in [20] tries to demonstrate the effectiveness of Urban Consolidation Centers for urban delivery by using multi-agents systems and reinforcement learning methods to evaluate the system under an uncertain environment of city logistics.

Finally, another important area in urban delivery where multi-agent systems and agent-based simulations can be useful is the simulation of interactions between retailers, drivers, and customers in a complex and dynamic scenario as is a city. In [21], authors propose an agent-based simulation in order to introduce a dynamic vehicle routing where the system tries to be adaptive to the freight demand. The main goal of the proposed system is to minimize the global traveled distance. The review published in [22] includes

more examples which try to demonstrate that agent-based simulations are suitable for modeling urban delivery distribution scenarios, but also evidences that the majority of the reviewed proposals are just test cases or theoretical models without a direct application to real systems.

In this paper, we propose a specific support tool to analyze and evaluate different models and strategies in the field of urban mobility and, specifically, in urban delivery scenarios. The developed tool is flexible enough to support several types of deliveries, transport companies, logistics, and many other mobility requirements. In addition, it allows for the configuration of different scenarios that reflect new modes of delivery such as collaborative delivery or car-sharing solutions.

3. SimFleet: A Simulation Tool for Open Fleets

SimFleet is a simulation tool which provides MAS researchers and learners with a convenient environment where to develop and test complex coordination and negotiation scenarios, in the context of a city containing one or more fleets of transportation vehicles. The tool can be used to simulate any kind of fleet where a group of vehicles transport *items* (goods or people) from one location to another within the city. Examples of fleets would be, among others, courier companies, taxi services, freight transport (by trucks), bike rental services, etc.

The tool has been built as a multi-agent system running on top of the SPADE platform [23], where the different actors in the fleet simulation (transportation vehicles and customers requesting items to be transported) are modeled as agents which can interact with each other by means of the SPADE communication facilities. The tool has been designed to hide most of the complexity of developing a multi-agent application by providing the user with three different interaction interfaces and a consistent internal architecture by which the tool can be easily adapted to the needs of a particular simulation scenario. The following subsections detail these main aspects of *SimFleet*, namely its three user interfaces, its underlying platform (SPADE), its internal architecture, and the way to incorporate custom strategies in order to adapt it to new fleet scenarios.

3.1. The User Interfaces

SimFleet provides the user with three different interfaces which can be selectively used to configure the tool in order to simulate several different fleet

scenarios: the Graphical User Interface (GUI), the command-line execution interface and the Application Program Interface (API). These interfaces are now described.

The GUI is the part of the application that runs the simulator where the transportation vehicles and items to be transported are displayed within the city map, and where the evolution of the simulation can be observed in real-time (it displays the initial location of the transportation vehicle and customer agents and their movements within the city as they interact with each other). This interface includes a limited set of interactions with the user. In particular, the GUI includes actions for starting and stopping the simulation and to zoom in or out in order to change the visible area, and outputs some basic simulation statistics. The GUI can be adapted to the needs of particular simulation scenarios by means of some features which are specified in the simulation configuration file (which is fully described below). First, the user can decide which city will be used for running the fleet simulation. Both the city map and the available routes within that city (the locations and directions of the streets) are based on open data which can be fed to the GUI and to the *Route Planner* agent, which calculates a valid route for every agent which needs to travel between any two given points within the city (this agent is discussed below in Section 3.3). This allows for running fleet simulations in any city where that open data is available, which virtually includes any important city in the world. And second, the GUI can be customized to represent each simulation agent (customer or vehicle) with its own particular icon, encoded in *Base 64* text format in the configuration file. For agents which do not specify their own icon, the GUI incorporates a library of default icons. In particular, there are different icons representing people which are randomly assigned to customer agents and a predefined icon for the most common types of vehicles (trucks, bikes, motorcycles, taxis, drones, electric vehicles, etc.); additionally, the GUI automatically changes the icon color of a vehicle if the simulation contains two fleets of vehicles of the same type. Figure 1 shows two screenshots of different simulation scenarios where these customizations can be observed: the top image shows a simulation of drone and motorcycle delivery in Madrid, while the bottom image shows a simulation of two cab companies in New York city.

The command-line interface is the way to specify all the parameters of a simulation to be executed with the GUI, as well as to run simulation offline (without the GUI). This is a very convenient way to extend the functionality of the simulation tool (in general, or for a particular fleet environment)

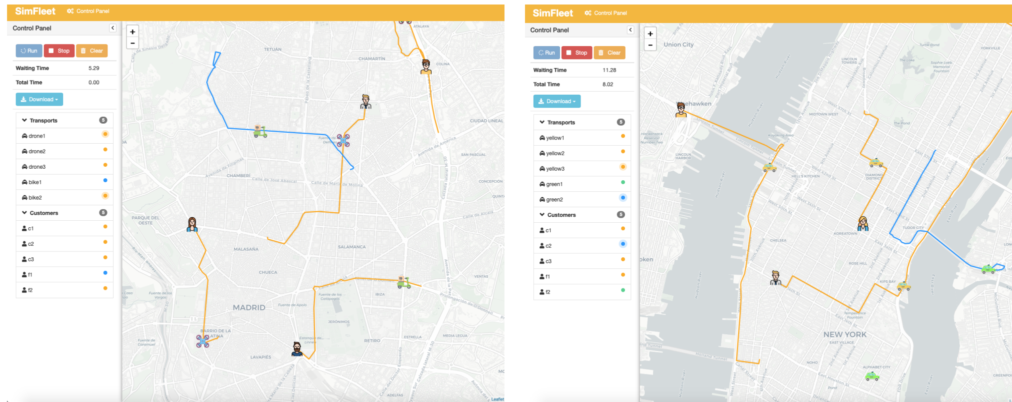

Figure 1: Screenshots of two simulations of SimFleet: motorcycle and drone delivery scenario in Madrid (left) and two cab companies scenario in New York City (right).

without having to modify the graphical interface. For the sake of simplicity, the command-line interface basically includes the loading of a configuration file which specifies, in a well-defined syntax, all the configurable aspects of a simulation scenario. Among others, relevant aspects of this file include the following:

- Fleet list. Each fleet is defined by specifying its fleet manager and, optionally, the strategy behavior of the manager and some fleet characteristics, including the fleet type (type of vehicle), along with the icon to represent these vehicles.

- Transportation vehicles list. This list includes the properties of every transportation vehicle to be included in the simulation: name, initial location in the map, speed (in km/h), fleet type, in which fleet(s) will be registered, personalized icon, and custom negotiation strategy.

- Customer list. This is the list of the customers that will interact with the transportation fleets in the simulation. For each customer, the following aspects can be defined: name, initial location in the maps, type of fleet service needed, final destination of the service, personalized icon and custom negotiation strategy.

- General simulation parameters. There are several parameters in this group, many of them configuring minor aspects of the application, such

as the name or duration of the simulation. Other parameters, more relevant to this description, would include the default negotiation strategies for each type of simulation agent (fleet, transportation vehicle, customer) to be used for any agent which does not specify its own custom strategy, the names of two relevant "system" agents (route agent and directory agents, introduced below), or the coordinates (latitude and longitude) of the city to be used in the simulation.

The third interface is the *SimFleet* API. When developing a new simulation scenario for a particular fleet (or set of fleets), the user typically needs to program the specific behaviors of the different simulation actors involved (vehicles, items, fleet managers, etc.), including their interactions and their respective negotiation strategies. In this context, some API functions are general, but many others may naturally be specific for the particular fleet environment that the user is developing (package delivery, taxi service, etc.). In order to facilitate this, the tool incorporates an abstract interface which can be easily extended in order to adapt the API to the fleet environment under study. This adaptation is twofold: on the one hand, the particular agent classes representing the simulation actors in a particular scenario can be derived from some abstract agent classes which already incorporate much of the common behaviors of such actors; hence, the user only has to focus on implementing the particular behavior of the scenario. On the other hand, in a particular scenario where the actual fleet actors (agents) have been implemented, *SimFleet* also incorporates some abstractions which assist in the development of different negotiation strategies among such agents. Some of these abstractions, such as a generic strategy behavior based on finite-state machines or the *Strategy Pattern* to dynamically incorporate strategies to agents without modifying the tool's code, are explained below in Section 3.3.

3.2. The SPADE Platform

SPADE (Smart Python multi-Agent Development Environment) is a multi-agent system (MAS) platform based on two main technologies: the XMPP[1] (eXtensible Messaging and Presence Protocol) standard for messaging and presence [24], and the Python programming language. These technologies offer many features and facilities that assist in the construction of MAS, such as an existing communication channel, the concepts of users (agents)

[1] http://xmpp.org

and servers (platforms) and an extensible communication protocol based on XML (eXtensible Markup Language).

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, such as IoT (Internet of Things), WebRTC[2] (Web Real-Time Communication), social, etc. SPADE itself uses some XMPP extensions to provide extended features to its agents, such as remote procedure calls between agents (Jabber-RPC[3]), file transfer (In-Band Bytestreams[4]), and so on.

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.

One of the most distinctive characteristics of SPADE as a multi-agent platform is the availability of an instant presence notification system, which is defined by the XMPP protocol, and provided by the XMPP server on which the platform is supported. This indirect communication mechanism is provided on a subscription basis, where the agents which are interested in the presence status of a particular agent can subscribe to it, and then be notified by the platform whenever that agent changes its status. Therefore, this notification system can be used by any SPADE agent in order to inform other agents about its own current, domain-dependent, status. For example, in the fleet domain, this system could be used in order to notify customers about the availability status of taxis or rental bikes, to inform a fleet manager whether each of its cargo trucks are in delivery or going to pick up packages, etc.

SPADE also provides a facility which helps the development (implementa-

[2]https://www.w3.org/TR/webrtc/
[3]https://xmpp.org/extensions/xep-0009.html
[4]https://xmpp.org/extensions/xep-0047.html

tion and debugging) of multi-agent applications from the perspective of each individual agent. In SPADE, each agent is provided with its own graphical interface, which is offered via web in a particular URL defined by the agent. The interface that SPADE provides by default to every agent contains a graphical representation of some internal information about the agent (including its message log, its active behaviors, its contact list, etc.) which can be consulted in real-time as the agent runs. In addition, SPADE allows agents to use this functionality in order to create other web graphical interfaces and to offer them as new URLs. In particular, SPADE proposes the Model-View-Controller (MVC) design pattern[5] for agents to define such interfaces. This is a very convenient way, for example, to develop the GUI of a multi-agent application in SPADE.

Finally, SPADE has been designed and implemented with scalability as a primary goal. In this sense, although the platform allows agents of the same multi-agent application to run in different processes on the same or different nodes over the internet, it also permits to efficiently execute several hundred agents within the same process in a single node. This is accomplished by implementing agents with Python's asynchronous programming library (called *AsyncIO*) and by optimizing the synchronization and communication facilities of SPADE for the case of agents residing in the same process.

3.3. The SimFleet *Architecture*

Internally, *SimFleet* is structured in four layers, which have been designed in order to separate the tool functionality and to make it easy to adapt it to particular fleet and/or negotiation and coordination scenarios. Figure 2 depicts this architecture, including the four layers (the simulator, the fleet, the agents, and their respective strategies), which are now described.

The simulator is the agent controlling the simulation process and serving the GUI. By means of the simulation configuration file, described above, it can be configured to locate the simulation display in a particular city. The simulator is also in charge of creating the rest of application agents in the simulation. Among these agents, there are two supporting agents which are also logically located in this layer: the *Route Planner* agent and the *Directory* agent.

The *Route Planner* agent is the agent in charge of calculating valid routes

[5]http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/mvc.html.

on the city streets for the vehicles traveling from one point to another within the city. This agent internally runs a routing engine provided by the Project OSRM[6] (Open Source Route Machine). It interacts with any agent requiring a valid route within the city (typically, the Transporter agents) by means of a particular performative called *"ROUTE"*, by which the requesting agent sends the origin and destination coordinates, and the Route Planner replies with the best of the valid routes which it can find between these points (in GeoJSON format), along with the route length and an estimate of the travel duration.

The *Directory* agent provides a typical directory service where agents providing services can register in order to be located by other agents which need these services dynamically during the simulation. In a typical SimFleet scenario, fleets register themselves in the *Directory* at the beginning of the simulation, and then any transporter vehicle which wants to enroll a fleet or any customer which requires a service of a particular fleet type must first locate the available Fleet Managers by asking this *Directory* agent. Some scenarios may have other services available (e.g., gas stations to refuel taxis, or docking stations for rental bikes) which would also use this agent.

The second layer contains the fleets which are required in any given simulation. In this context, the Fleet abstraction represents a company or institution which owns a group of vehicles which are capable of transporting goods or people from one point to another within the city. Each vehicle in the fleet has its own agent in the simulation, called the *Transporter*, described below in the agent layer. In addition to the transporter agents, each fleet includes another agent, called the *Fleet Manager*, which is in charge of receiving the requests from the customers and (potentially) making some decisions about how to organize the requests and to coordinate the transport process of the fleet's vehicles. The amount of coordination and decision making of the manager depends on the strategies of both the manager and the transporter agents, allowing for the simulation of both centralized and decentralized fleets. As explained above, the *Fleet Manager* agents use the *Directory* agent in order to register their respective fleet services, hence allowing the Transporters and the Customers to locate these services.

At the agent layer, the tool includes an agent for the Fleet Manager and all the Transporters of each fleet, and also an agent for every Customer which

[6]http://project-osrm.org

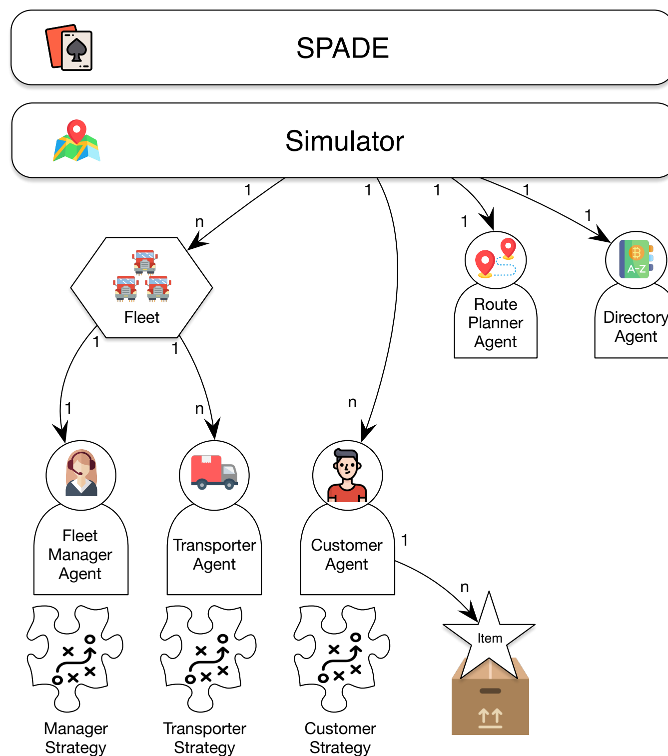

Figure 2: The SimFleet Architecture.

may issue transport requests in the simulation. A Transporter agent represents both a vehicle and its driver, and it has some attributes which may be customized: maximum speed, cost per kilometer (or per delivery), fleet, name of the fleet manager, initial position, and its icon (in the simulator display). On the other hand, a Customer is an agent which may request transport services for any available fleet, and which includes some attributes such as the amount of money available, the list of items to be transported or its personalized icon in the simulation. Customer agents request transport services by sending messages to one or many Fleet Managers, the addresses of which are obtained first by consulting the *Directory* agent, as explained above. It is worth noting that the items to be transported are not represented as agents in *SimFleet* since all the negotiation processes are carried out by the item's owner, the Customer agent. For this reason, items are here modeled as *artifacts* including two main attributes, its location coordinates and its display icon. This model also allows for the case where the item to

be transported is also the customer (for example, when a customer requests a taxi service), by linking the customer's location to the item's location.

The fourth layer comprises the coordination and negotiation strategies that the simulation agents apply in order to request, select, organize and coordinate the transport services. Each agent incorporates its own strategy, implemented as a SPADE behavior, according to its role in the simulation environment (Customer, Transporter or Fleet Manager) and its own particular goals. The tool provides a basic, default strategy for each of these three roles, which can be changed for each fleet scenario; but even in the same fleet scenario, it is also possible to define a particular strategy of each individual agent, hence allowing for much more sophisticated simulations. This layer has been designed by using the so-called *Strategy Pattern* [25], which is the best practice when an application incorporates different alternative versions of an algorithm and the user wants to be able to select for execution any of these versions at run time. *SimFleet* uses this design pattern in order to dynamically attach new, custom negotiation behaviors for the simulation agents (customers, transporters, managers) at run time. In particular, the files containing the code of those behaviors can be specified at the simulation configuration file when a simulation is launched, and therefore there is no need to modify the tool's code files. The next section describes how to implement the code of these behaviors.

3.4. Developing Custom Strategies with SimFleet

The *SimFleet* architecture provides a well-defined API in order to define the negotiation strategies of the simulation agents. In particular, each agent includes a specific SPADE behavior (defined in Section 3.2), which implements the code establishing the negotiation goals and interactions with the relevant counterpart agents. The tool provides two alternative ways to implement such strategy behaviors, depending on their complexity and the developer preferences. Specifically, strategies can either be built by using a basic, cyclic behavior (inherited from the *SimFleet* `StrategyBehaviour` class) or they can be internally structured as finite-state machines (inherited from the *SimFleet* `FSMStrategyBehaviour` class). In this latter case, the strategy is implemented in a way in which each state represents a possible situation in the agent's negotiation strategy (and implements the actions to be executed in that situation), and each transition represents a possible change from one situation to another within the strategy. The `FSMStrategyBehaviour` is best suited for complex negotiation scenarios in which embedding the entire

strategy into a single (`StrategyBehaviour`) class could be difficult. This way, the tool offers a consistent framework for designing and developing negotiation strategies of different complexity levels, where the actual code to be implemented is very little.

For the sake of simplicity, this section will focus on the `StrategyBehaviour` case. Following the strategy pattern described in the previous section, when defining a new fleet scenario, developers need only to implement three classes provided by *SimFleet* which define the strategies of the three simulation agent types: the fleet manager's `FleetManagerStrategyBehaviour`, the customer's `CustomerStrategyBehaviour` and the transporter's `TransportStrategyBehaviour`. These are cyclic SPADE behaviors, inherited from the *SimFleet* default `StrategyBehaviour` class, that run in an infinite loop until the agent stops. Each one has a coroutine called `async def run(self)` where the code that defines the strategy behavior of the agent must be placed. An example of this is shown in Listing 1.

```
1  from simfleet.fleetmanager import
       FleetManagerStrategyBehaviour
2  from simfleet.customer import CustomerStrategyBehaviour
3  from simfleet.transport import TransportStrategyBehaviour
4
5  class MyFleetManagerStrategy(FleetManagerStrategyBehaviour):
6      async def run(self):
7          # Your code here
8
9  class MyTransportStrategy(TransportStrategyBehaviour):
10     async def run(self):
11         # Your code here
12
13 class MyCustomerStrategy(CustomerStrategyBehaviour):
14     async def run(self):
15         # Your code here
```

Code Listing 1: An example of the strategy development in SimFleet.

Developers have plenty of freedom when implementing such behaviors. They can use any intelligent algorithm, send and receive messages by using the SPADE communication facilities, or even create additional agent behaviors if needed. In addition, *SimFleet* provides a set of tools and helpers that facilitate the development of the strategies. Among others, there are methods which facilitate interactions related to the transport domain (like `accept_transport(transport_id)`, `refuse_transport(transport_id)` or

`send_request(transport_id)`), and some which deal with well-known problems related to working with geo-located data (like `are_close(coord1, coord2, tolerance)`, which confirms whether two coordinates are close within a particular tolerance range, or `distance_in_meters(coord1, coord2)`, which computes the distance between two coordinates).

Overall, these development facilities allow for the creation of a wide range of fleet scenario simulations located in different cities around the world, and to consistently implement and test alternative negotiation and coordination strategies in such scenarios.

4. Case of Example: Last Mile Delivery

This section presents the application of *SimFleet* to the so-called Last Mile Delivery (LMD) problem [26]. The LMD is the main problem which needs to be dealt by logistics and distribution processes in a city, that is, to plan the delivery of items to their final destination within the city. In this sense, the adaptation of *SimFleet* to this problem will include the agent roles and their interaction protocol, along with the way agents behave and the different strategies used to assign Customer agents (with a parcel to be sent) to Transporter agents. The rest of the section is devoted to explain such adaptations, which will show how *SimFleet* is suited for a complex case of use like the one described here.

There are different approaches which may tackle the LMD problem. This section explains first how *SimFleet* could be used to simulate three of them (named *Traditional Fleets*, *Open Fleets* and *Crowdsourcing Fleets*) individually, and then presents a simulation scenario where these three alternatives are combined in order to present a solution for this problem.

In a *Traditional Fleet* scenario, each logistic fleet comprises a set of Transporter Agents and has a Fleet Manager agent associated to it. Each Customer Agent with a parcel to be sent will issue a delivery service request to the Fleet Manager agent, providing a pickup point and a final destination for the parcel. Then, the Fleet Manager will assign a Transporter Agent from its own logistics company (or fleet) to take the parcel and carry it to that final destination. Figure 1-left shows an example of such simulation scenario using *SimFleet*, in this case with two logistic fleets.

Open Fleets present an specially interesting case in Logistics Fleets. In an open fleet scenario, Transporter Agents do not belong to a logistic fleet, but on the contrary, they may enter and leave any of the logistics fleets at

will, whenever they want. In this case, Transporter Agents are paid by the parcels they deliver, which as usual are requested by Customer Agents. This approach can be easily simulated in SimFleet, as any new Transporter Agent may ask the Directory Agent for the identifiers of the Fleet Managers, and then send messages to such Fleet Manager agents to enroll the fleet that they prefer.

The third basic type of fleet described here is based on a *crowdsourcing* approach. One of the important aspects in LMD logistics is the cost associated to these delivery operations. This cost includes, but it is not limited to, the cost related to sustainability. That is, all the transportation vehicles which are moving through the city have an direct impact in the city pollution. In [26], there is a new approach to LMD problem which considers such sustainability cost, based on a crowdsourcing solution. In a Crowdsourcing Fleet scenario, the solution would be based on an open fleet with *temporary* Transporter Agents, where such Transporter Agents do not take a specific route to deliver each parcel from the Customer's pickup point to the final destination, but on the contrary, they make use of their usual routes in order to carry a parcel, either to its final destination or to a point where another Transporter Agent can pick it up and continue with the delivery process.

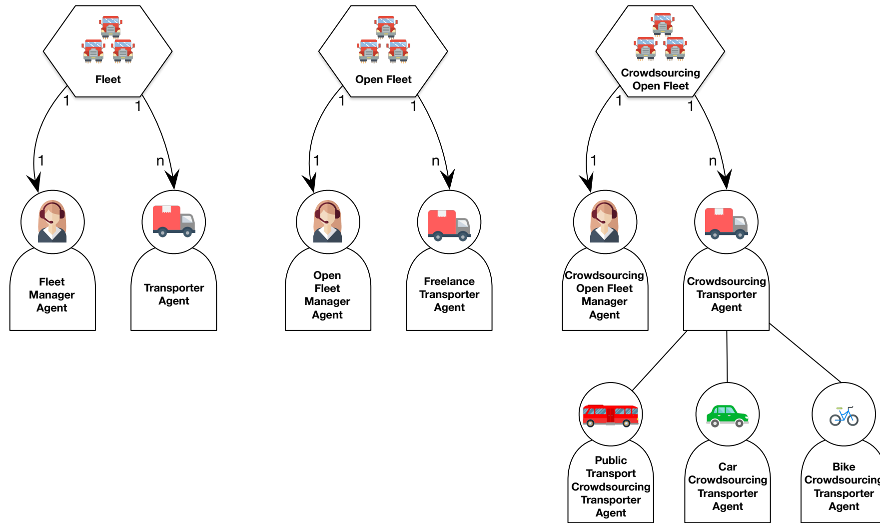

Figure 3: Fleet taxonomy in the Last Mile Delivery example.

With all this, we have made a simulation to test a solution of the LMD problem which uses such Crowdsourcing Fleets, where we have a Crowd-

sourcing Open Fleet, a normal Open Fleet and a Traditional Fleet located in the city of London. Figure 3 shows the taxonomy of the different fleets used in the example. Each one of these fleets has its own Fleet Manager, so there is a *Crowdsourcing Open Fleet Manager*, an *Open Fleet Manager* and a *Fleet Manager*. Also, there are three different types of *Transporter Agents*:

- *Transporter Agents*: the ones modeling traditional transporters who work for a fleet company as a full-time job. The movements through the city of these transporters are always in order to carry parcels from their pickup points to their destinations.

- *Freelance Transporter Agents*: the ones modeling transporters that may accept delivery services from an Open Fleet company for some time (or number of deliveries), and after that they "leave" the company.

- *Crowdsourcing Transporter Agents*: the ones modeling people that may enter a Crowdsourcing Open Fleet through a mobile app, by giving their usual routes and availability. While they are logged on in the app, they can accept to be assigned parcels to be carried along their usual routes, typically if they do not have to modify that route significantly. In this simulation, there will be different Crowdsourcing Transporter Agents according to the transport vehicle they use: car, bike or public transport.

When the Crowdsourcing Fleet Manager receives a message from a Customer Agent requesting to carry a parcel from an origin location to a destination point, the following process is carried out:

- It tries to make a valid path using the routes of the Crowdsourcing Transporter Agents currently logged in this fleet. The idea is to use these routes as much as possible, having the best solution if the parcel can be carried by only a single Crowdsourcing Transporter Agent, but otherwise also being a valid solution if the transport can be made by combining the routes of several of these agents.

- If it is not possible to use only those agents for the transport, the Crowdsourcing Fleet Manager would try to contract a *Freelance Transporter Agent* to contract with this second fleet the part(s) of the route not covered by Crowdsourcing Transporter Agents in the previous step.

- If there are still parts of the delivery route for the parcel which are not covered, the Crowdsourcing Fleet Manager attempts to subcontract a traditional fleet company to carry the parcel over such route parts. This is done by negotiating with the Fleet Managers of such fleets, by using a First-price sealed-bid auction. In fact, there will be one auction for each non-covered route parts. So, the Crowdsourcing Fleet Manager sends a Call-For-Proposals message to the active Fleet Managers in the system with the origin and destination positions of this non-covered part, along with the estimated arrival time of the parcel to the part's initial position. The Fleet Managers will send their proposals, composed of a cost and the estimated time for the parcel to arrive to the destination point. In each case, the Crowdsourcing Manager will resolve the auction and will answer the winning Fleet Manager with an agree message (and cancel the others).

Figure 4 shows an snapshot of the simulation presented above, where there is a Crowdsourcing Open Fleet that has sent a parcel to be carried out by three different Transporter Agents, being the first two ones Crowdsourcing Transporter Agents, and the last one a traditional Transporter Agent, working for a traditional Fleet Manager that has been subcontracted by the Crowdsourcing Open Fleet Manager. In the Figure you can see a blue line marking the bike Transporter Agent route, a red line marking the public transport Transporter Agent route, and a yellow line marking the route of the the traditional Transporter Agent. It can also be observed, in dotted lines, the delivery route followed by the parcel, using the usual routes of the two first Transporter Agents and the ad-hoc route of the traditional Transporter Agent for the last part of the delivery.

This example has illustrated how to adapt *SimFleet* to a simulation scenario with some specific fleet types (in this case a traditional fleet, an open fleet and a crowdsourcing open fleet) and which elements would be necessary to design. Once the tool has been adapted to such scenario, many different strategies for each agent type (Fleet Manager, Transporter, Customer) could be implemented and tested in any number of simulations.

5. Conclusions

Urban mobility has changed dramatically in recent years with the emergence of new modes of transport and with technological advances. In this

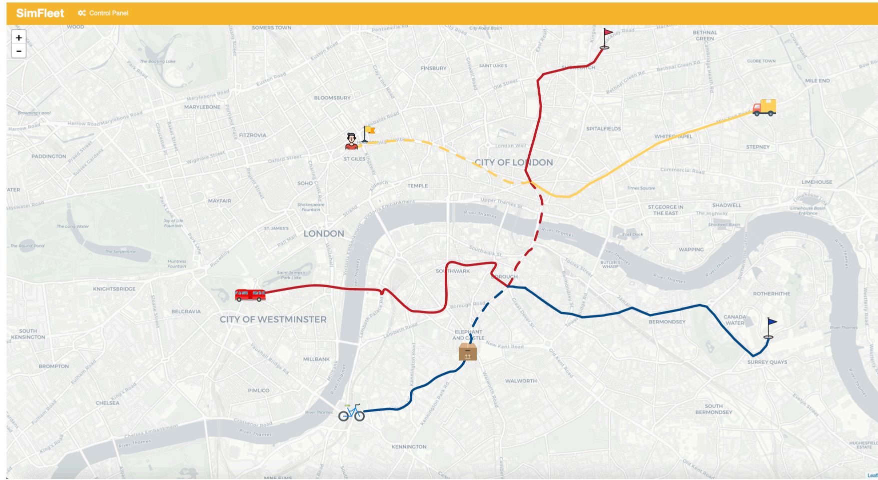

Figure 4: Crowdsourcing Open Fleet example.

sense, the evaluation of new models or strategies is very complex since it is difficult to determine in advance their adequate performance and their possible improvements against other alternatives. It is for this reason that this paper presents a tool called *SimFleet* which has been specifically designed to simulate and evaluate new models of urban mobility in current cities. The main goal of *SimFleet* is to allow users for the development and test of new models or policies of mobility management in the urban environment, as well as to analyze possible new coordination strategies and regulatory mechanisms of vehicle fleets that can improve the efficiency in the distribution of people or goods in urban environments.

The *SimFleet* tool has been designed and implemented over the SPADE platform, which allows the development of multi-agent systems in Python. In this way, the simulation is designed by generating an agent for each of the different actors involved in the process (vehicles, passengers, parcels, managers, etc.). The tool has been designed to facilitate the aggregation of new models and strategies to the agents that are running in the simulation, so that agents can dynamically change their strategies and adapt to the context. On the other hand, *SimFleet* allows for the definition of different simulation scenarios and, also, the definition of appropriate metrics to analyze and compare the different strategies, models or policies that have been defined.

Finally, the paper illustrates the use of the tool with a proof of concept that simulates a crowdsourcing approach to address the LMD problem by

using fleets of different types. As future work, the tool will incorporate the concept of artifact in order to model static entities into the simulation, such as elements of the city infrastructure (traffic lights, sensors, etc.).

Acknowledgment

This work was partially supported by MINECO/FEDER RTI2018-095390-B-C31 of the Spanish government.

References

[1] H. Billhardt, A. Fernández, M. Lujak, S. Ossowski, V. Julián, J. F. De Paz, J. Z. Hernández, Towards Smart Open Dynamic Fleets, in: Multi-Agent Systems and Agreement Technologies, Springer, 2015, pp. 410–424 (2015).

[2] O. S. Lujak M., Giordani S., Route guidance: Bridging System and User Optimization in Traffic Assignment, Neurocomputing 151 (1) (2015) 449–460 (2015).

[3] E. Adam, E. G.-L. Strugeon, R. Mandiau, MAS architecture and knowledge model for vehicles data communication, ADCAIJ: Advances in Distributed Computing and Artificial Intelligence 1 (1) (2012) 23–31 (2012).

[4] P. Chamoso, F. de la Prieta, Swarm-Based Smart City Platform: A Traffic Application, ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal 4 (2) (2015) 89–97 (2015).

[5] A. F. Isabel, R. F. Fernandez, Simulation of Road Traffic Applying Model-Driven Engineering, ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal 4 (2) (2015) 1–24 (2015).

[6] L. Aboueljinane, E. Sahin, Z. Jemai, A review on simulation models applied to emergency medical service operations, Computers & Industrial Engineering 66 (4) (2013) 734–750 (2013).

[7] R. Nair, E. Miller-Hooks, R. C. Hampshire, A. Bušić, Large-scale vehicle sharing systems: analysis of vélib', International Journal of Sustainable Transportation 7 (1) (2013) 85–106 (2013).

[8] C. M. Macal, M. J. North, Tutorial on agent-based modelling and simulation, Journal of Simulation 4 (3) (2010) 151–162 (Sep 2010).

[9] J. Palanca, A. Terrasa, C. Carrascosa, V. Julián, SimFleet: A New Transport Fleet Simulator Based on MAS, in: International Conference on Practical Applications of Agents and Multi-Agent Systems, Springer, 2019, pp. 257–264 (2019).

[10] G. Gentile, K. Noekel, Modelling public transport passenger flows in the era of intelligent transport systems, Gewerbestrasse: Springer International Publishing (2016).

[11] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, D. Rus, On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment, Proceedings of the National Academy of Sciences 114 (3) (2017) 462–467 (2017).

[12] P. Neirotti, A. D. Marco, A. C. Cagliano, G. Mangano, F. Scorrano, Current trends in Smart City initiatives: Some stylised facts, Cities 38 (2014) 25 – 36 (2014).

[13] D. Krajzewicz, J. Erdmann, M. Behrisch, L. Bieker, Recent development and applications of sumo-simulation of urban mobility, International Journal On Advances in Systems and Measurements 5 (3&4) (2012).

[14] A. Horni, K. Nagel, K. W. Axhausen, The multi-agent transport simulation MATSim, Ubiquity Press London, 2016 (2016). `doi:10.5334/baw`.

[15] C. Rajeshwari, Optimizing last mile delivery using public transport with multiagent based control, Master's thesis, LUT University (2016).

[16] C. Chen, S. Pan, Using the crowd of taxis to last mile delivery in e-commerce: a methodological research, in: Service Orientation in Holonic and Multi-Agent Manufacturing, Springer, 2016, pp. 61–70 (2016).

[17] H. Paloheimo, M. Lettenmeier, H. Waris, Transport reduction by crowdsourced deliveries – a library case in Finland, Journal of Cleaner Production 132 (2016) 240 – 251 (2016).

[18] O. Wangapisit, E. Taniguchi, J. S. Teo, A. G. Qureshi, Multi-agent systems modelling for evaluating joint delivery systems, Procedia-Social and Behavioral Sciences 125 (2014) 472–483 (2014).

[19] R. Alves, R. da Silva Lima, D. Custódio de Sena, A. Ferreira de Pinho, J. Holguín-Veras, Agent-based simulation model for evaluating urban freight policy to e-commerce, Sustainability 11 (15) (2019).

[20] N. Firdausiyah, E. Taniguchi, A. Qureshi, Modeling city logistics using adaptive dynamic programming based multi-agent simulation, Transportation Research Part E: Logistics and Transportation Review 125 (2019) 74–96 (2019).

[21] B. M. Sopha, A. Siagian, A. M. S. Asih, Simulating dynamic vehicle routing problem using agent-based modeling and simulation, in: 2016 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), IEEE, 2016, pp. 1335–1339 (2016).

[22] A. Nuzzolo, L. Persia, A. Polimeni, Agent-based simulation of urban goods distribution: A literature review, Transportation research procedia 30 (2018) 33–42 (2018).

[23] M. Escrivà, J. Palanca, G. Aranda, A jabber-based multi-agent system platform, in: Proceedings of the 5th int. joint conference on Autonomous agents and multiagent systems, ACM, 2006, pp. 1282–1284 (2006).

[24] P. Saint-Andre, Extensible messaging and presence protocol (XMPP): Core, RFC 6120 (2011).
URL `https://tools.ietf.org/html/rfc6120`

[25] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA., 1995 (1995).

[26] A. Giret, C. Carrascosa, V. Julian, M. Rebollo, V. Botti, A crowdsourcing approach for sustainable last mile delivery, Sustainability 10 (12) (2018).