Micro City - Simulation Report

Alessandro Marcantoni
Simone Romagnoli

August 2, 2022

Contents

1 Introduction

Usually, in order to analyze and understand a system, the traditional scientific method suggests observing a phenomena and possibly measuring it under controlled conditions. This workflow allows the observers to formulate hypothesis and then either validate them or refute them. Nevertheless, it is not always possible to observe a system either for practical or ethical reasons or maybe because the system is too complex, or it does not exist. This is the reason why **simulation** is introduced as a new way for describing scientific theories. It is defined as the process with which we can study the dynamic evolution of a model system, usually through computational tools [1].

As already discussed in the case-study-analysis, the aim of this study is to determine whether a *situated recommendation system* can positively affect the visitor flow in an amusement park or not. Since the desired system does not exist, and it would be too costly to implement just for the sake of a proof of concept, the only way to validate the latter theory is through a simulation.

Building a simulation also allows working with a *model* which is a representation and abstraction of the actual system. This also means that the model will be simpler and will only contain those aspects that are strictly essential for the sake of the study that will be made.

2 The Case Study

From now on, the terms of the *ubiquitous language* identified in the analysis document will be used to describe the desired model; please, refer to case-study-analysis to avoid ambiguities.

As already discussed in the above analysis, attractions may cause considerable queues with long waiting times. The aim of this simulation is to determine whether a *situated recommendation system* may help reduce the waiting time needed to benefit from an attraction. In order to achieve this, two simulations will be developed:

1. **Random Redirection**: this one should reflect the current system, that is, once a visitor is satisfied by an attraction they move towards another attraction which is statistically chosen randomly.

2. **Recommended Redirection**: this one should reflect the desired system, that is, once a visitor is satisfied by an attraction they move towards the next one accepting a recommendation. The latter may suggest an attraction if it has a short queue and if it is close enough to the visitor.

Specifically, the model of the two simulations will include the following aspects:

- A real world map of the amusement park, that is *Mirabilandia*, one of the most famous amusement parks in Italy. In particular, the simulation will only focus on the environment inside the park's boundaries. Moreover, the map will contain a network of routes that connect the attractions and can be exploited by visitors to move around the park.

- Attractions of many types (rides, water slides, restaurants, etc.) physically situated on the map. Their location will adhere to the actual location inside the park. Finally, they will be able to dequeue a specific amount of visitors, that had previously enqueued, and satisfy them.

- Visitors that will be able to move towards a specific attraction, chosen with one of the above policies, and wait for their turn in order to benefit from the attraction itself. Once they have been satisfied, they will proactively choose the next attraction.

3 The Alchemist Metamodel

As for the implementation of the simulation, the *Alchemist Simulator* [2] was used for the following reasons:

- It provides a *ready-to-use* simulation environment, avoiding to implement one from scratch.
- It provides an intuitive graphic user interface that allows observing the evolution of the developed simulation.
- To contribute to the *research and development* area of the *Dipartimento di Informatica - Scienze e Ingegneria* (*DISI*) of the *University of Bologna*.

Alchemist presents its own metamodel, providing a series of concepts necessary to be understood in order to develop a simulation. In this section, only the crucial aspects will be reported, in order to fully understand the current simulation; for further information, please check the official documentation. The main entities of the metamodel are:

- **Molecule**: the name of a data item; it can be seen as a variable name.
- **Concentration**: the value associated to a specific molecule.
- **Node**: a container of molecules and reactions.
- **Environment**: the space abstraction that contains nodes and provides them with spatial capabilities (position, movement, etc.).
- **Linking Rule**: a function of the environment that associates a node with others.
- **Reaction**: an event that can change the status of the environment.

The image 1 provides a visual representation of such metamodel:

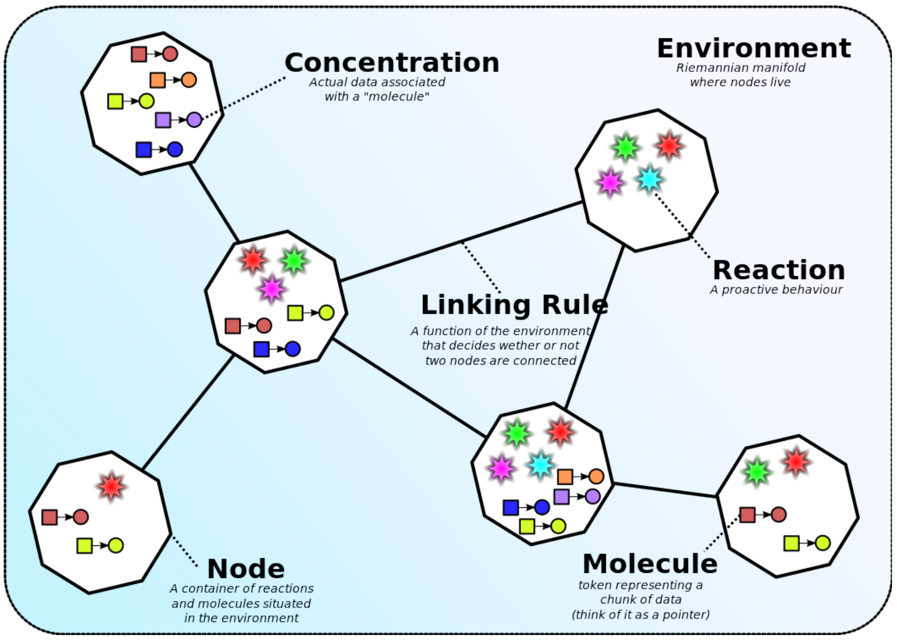
Concentration
Actual data associated with a "molecule"
Environment
Riemannian manifold where nodes live
Reaction
A proactive behaviour
Linking Rule
A function of the environment that decides wether or not two nodes are connected
Node
A container of reactions and molecules situated in the environment
Molecule
token representing a chunk of data (think of it as a pointer)

Figure 1: Visual representation of the *Alchemist* metamodel.

The *Alchemist* metamodel is implemented by different **incarnations**, that is to say, concrete instances of the metamodel. Among the different incarnations, the Protelis Incarnation was chosen, as it seems to be the one that better adapts to the current simulation. *Protelis* is a domain-specific language for *aggregate programming*. *Aggregate programming* produces reliable and robust collective behavior from uncoordinated local interactions between nodes, and provides higher-level abstractions for device-to-device communication and aggregate-level operations for distributed coordination [3].

4 The Simulation

In this section, we discuss the main elements of the simulation. To fully understand the concepts that are discussed here, such as nodes and programs, please refer to the Alchemist documentation.

4.1 Map Environment

The simulation concerns an existing geographical location, that is the amusement park of *Mirabilandia*. For this reason, the environment of the simulation must be a map featuring existing paths in the real world. To achieve this, *Alchemist* allows using maps provided by *OpenStreetMap*, the free wiki world map. *OpenStreetMap* provides navigation capabilities on the whole planet; such data, weighs about 50GB, thus it is recommended to use an extract with the data relative to the interested area. One great way to obtain an extract is through *BBBike* [2].

Once extracted the map in the `.pbf` format, it is possible to build the simulation environment through the `OSMEnvironment`, as shown in the following listing (1).

Listing 1: Building an *OpenStreetMap* environment.

```yaml
# Use an OpenStreetMap Environment,
# deploying nodes only on streets.
environment:
  type: OSMEnvironment
  parameters: [mirabilandia.osm.pbf, true]
```

The constructor of the environment accepts two parameters:

- `file: String`, the path to the file containing the exported map;
- `onlyOnStreets: Boolean`, a boolean value allowing to deploy nodes only on the streets.

4.2 Deployed Nodes

After setting the environment, it is essential to deploy **nodes** on the map. In the current simulation, the elements that are represented by nodes are:

- **Visitors**, as single individuals or groups; the key point here is that a node should correspond to one or more people using a single wearable device that tracks and guides its owner.
- **Attractions**, that can be of different types, such as rides, water slides, restaurants, etc.; they are considered *rendezvous* points for visitors and are made of several sensors that allow keeping track of various information, such as the number of people waiting in a queue.

In order to deploy nodes on the map it is mandatory to declare them under the `deployments` section, as shown in the following listing.

Listing 2: Deploying 1 attraction and 100 visitors inside the `bounds` polygon.

```
# Define visitors
_visitors: &visitors
  - type: Polygon
    parameters: [ 100, *bounds ]
    contents:
      - { molecule: visitor, concentration: true }

# Define attractions
_attractions: &attractions
  - type: Point
    parameters: [44.33589, 12.26293]
    contents:
      - { molecule: attraction, concentration: true }
      - { molecule: attractionType, concentration: "\"restaurant\"" }
      - { molecule: capacity, concentration: 10 }
      - { molecule: name, concentration: "\"McDonald's\"" }

# Deploy nodes
deployments:
  - *attractions
  - *visitors
```

Moreover, it is necessary to explicit which **linking rule** will be used to connect nodes with each other. As for the current simulation, the proper way to connect nodes cannot be based on a geometric rule (for instance, connecting nodes within a certain distance). Instead, it is appropriate to consider attractions as **access points** for the visitors' devices. Even with the `ConnectToAccessPoint` linking rule every node on the map will be connected with the rest of the network as the attractions are distributed throughout the map, and, working as access points, they can cover it without leaving connectionless areas.

Listing 3: Defining the linking rule: only nodes with the molecule `attraction` will connect to other nodes within a radius of 100 meters.

```
# The network model used allows to choose the nodes that have the
# "attraction" molecule in order to simulate an access point behaviour
network-model:
  type: ConnectToAccessPoint
  parameters: [100.0, "attraction"]
```

4.3 Programmed Behaviours

With both the environment and the nodes correctly set, the next step consists in programming their behaviours. Each and every node needs to implement a proper behaviour depending on its type. In particular, visitors and attractions will have different behaviours: for instance, visitors should move on the map in order to get to attractions, while the latter should stay still and satisfy enqueued visitors. The behaviours are written using the *Protelis* aggregate language. Although, performing local computations using a programming language designed for the aggregate paradigm is probably not the best choice. So, it is appropriate to implement local computations with a different language. Therefore, for this simulation, also *Kotlin* was used. The following sections describe the behaviours implemented by visitors and attractions.

4.3.1 Attractions' Positions

In order to make each visitor node aware of the positions of all the attractions in the map, it is necessary to implement a behaviour that spreads the desired information. To do so, it is useful to use *aggregate programming*. Specifically, *Alchemist* allows inserting an external program into a node, as shown in the following listing.

Listing 4: Assign the `positions` behaviour to 100 visitors and 2 attractions.

```yaml
# Define the broadcast message from attractions to visitors
# to spread their positions
_positions: &positions
  - time-distribution: 1
    program: org:protelis:microcity:positions
  - program: send

# Define visitors
_visitors: &visitors
  - type: Polygon
    parameters: [ 100, *bounds ]
    programs: *positions
    contents:
      - { molecule: visitor, concentration: true }

# Define attractions
_attractions: &attractions
  - type: Point
    parameters: [44.33747, 12.26208]
    programs: *positions
    contents:
      - { molecule: attraction, concentration: true }
  - type: Point
    parameters: [ 44.33815, 12.2633 ]
    programs: *positions
    contents:
      - { molecule: attraction, concentration: true }
```

On the other hand, the definition of the aggregate program should describe how the desired information is spread as a field. In particular, each node should gather the field from its neighbours and merge the received values. The listing 5 shows how it is possible to implement this behaviour in *Protelis*.

Listing 5: Sharing the coordinates of each attraction to all the nodes that implement this behaviour.

```
module org:protelis:microcity:positions

import microcity.Positions.attractionPositions
import microcity.Positions.attractionUnion

share (field <- attractionPositions()) {
    foldHoodPlusSelf(field, { a, b -> attractionUnion(a, b) })
}
```

The `attractionPositions` function simply builds a list with the node's position if it presents an "attraction" molecule, otherwise an empty list. In this way, the field is composed by lists of positions containing only attractions' coordinates. The `attractionUnion` function just unifies the received lists into a single one.

4.3.2 Queues & Satisfaction

Similarly to positions, it is necessary to keep track of the queues that form nearby attractions and broadcast them to visitors. The queue formed nearby an attraction is determined by the sequence of visitors that have the same coordinates as the attraction. Though, it is important that the visitors waiting in the queue are not satisfied. In fact, a visitor's satisfaction is a condition that only occurs after it has benefited from an attraction. When satisfied, the visitor is then ready to choose the next destination, and, once the next attraction is decided, it can switch back to unsatisfied. These behaviours can be split into 3 different *Protelis* programs:

- `queue`: given an attraction, determine the list of visitors that are enqueued to it.
- `queues`: broadcast the queues of every attraction to every visitor.
- `satisfaction`: given an attraction, satisfy the first N visitors in its queue, where N represents the attraction's capacity.

All these behaviours are programmed by both attractions and visitors. The `satisfaction` program has a lower time distribution, as it is assumed that it takes a while for an attraction to satisfy visitors.

4.3.3 Movement

Movement is a feature only owned by visitors. It is implemented as a `TargetMapWalker`: this action needs a "tracking" molecule inside the implementing node that has a destination's coordinates as its concentration. Moreover, the latter allows nodes to move only on maps' streets, adapting to the provided `OSMEnvironment`.

As discussed in the section 2, in order to decide its next destination, a visitor can adopt one of these two policies: **Random Redirection** or **Recommended Redirection**. In order to achieve this, visitors also own a program that establishes the next destination with one of the two policies and inserts its coordinates in a molecule. The listing 6 shows how the `TargetMapWalker` behaviour and the `destination` behaviour are assigned to the visitor nodes.

Listing 6: Define the movement behaviour for visitors.

```
# Define the movement law with a Walker that moves only on streets towards
# a GeoPosition indicated by the concentration of the tracking molecule
_move: &move
  - time-distribution: 1
    type: Event
    actions:
      - type: TargetMapWalker
        parameters: [org:protelis:microcity:destination, 1.0]

# Define the rule that establishes how visitors choose the
# next destination
_destination: &destination
  - time-distribution: 1
    program: org:protelis:microcity:destination
  - program: send
```

In order to choose the next destination for a visitor, a *Protelis* program will call a function `getNext` that will return the coordinates of the next attraction. It is possible to define a custom policy just by implementing the interface `NextPolicy`, and therefore defining the method `getNext`. For the sake of the current simulations, the following policies were provided:

- `RandomPolicy`: performs a *random redirection* by choosing one of the attractions inside the park randomly.
- `ShortestQueuePolicy`: performs a *recommended redirection* by choosing the attraction with the shortest queue inside the whole park.

- `ShortestQueueInRangePolicy`: performs a *recommended redirection* by choosing the attraction with the shortest queue within a given range with regard to the visitor.

4.4 Data Extraction & Evaluation

The whole point of the simulations consists in determining whether a *situated recommendation system* may help reduce the waiting time for visitors inside the park. In order to evaluate that, it is necessary to extract data from the simulation. In particular, the data we extracted and analyzed concerns the amount of time spent in a queue and the number of attractions that every visitor has benefited from. Data extraction is achieved thanks to the implementation of an `Extractor` that allows controlling and exploring the environment at each simulation step. Thanks to that, it was possible extract the waiting time that every visitor spends in a queue at each simulation step. On the other hand, counting the amount of times that a visitor has been satisfied by an attraction was possible thanks to a specific molecule `satisfactions`. These two types of data, on average, considering both the *random redirection* and the *recommended redirection* systems, consent to state whether visitors benefit from more attractions and wait less time in queues or not; thus, they are considered to be a reasonable way to evaluate the *situated recommendation system*.

For the evaluation, the right way to obtain reliable data is to launch every simulation multiple times and finally consider the average values of such data. The following evaluations are based on the data obtained from 10 simulation runs for each redirection policy described in the section 4.3.3.

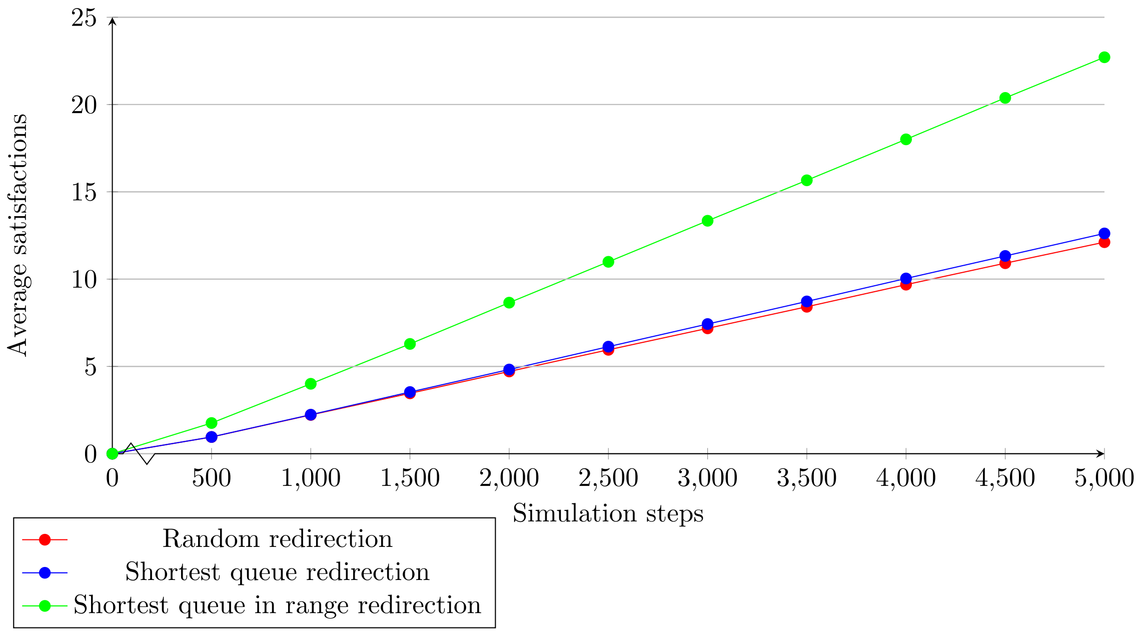

Figure 2: Average satisfactions for the three different redirection policies.

As shown in the plot 2, the number of satisfactions increases with the simulation steps. The *random redirection* is clearly the worst case, as visitors could cross the entire map in order to reach the next attraction. A light improvement is given by the *shortest queue redirection*, that allows visitors to reach attractions with shorter queues and, therefore, benefit from more of them. Instead, *shortest queue in range redirection* ensures that the next attraction will be within a certain range. In this way visitors will walk less in order to reach attractions. In fact, this policy brings on a higher number of satisfactions.

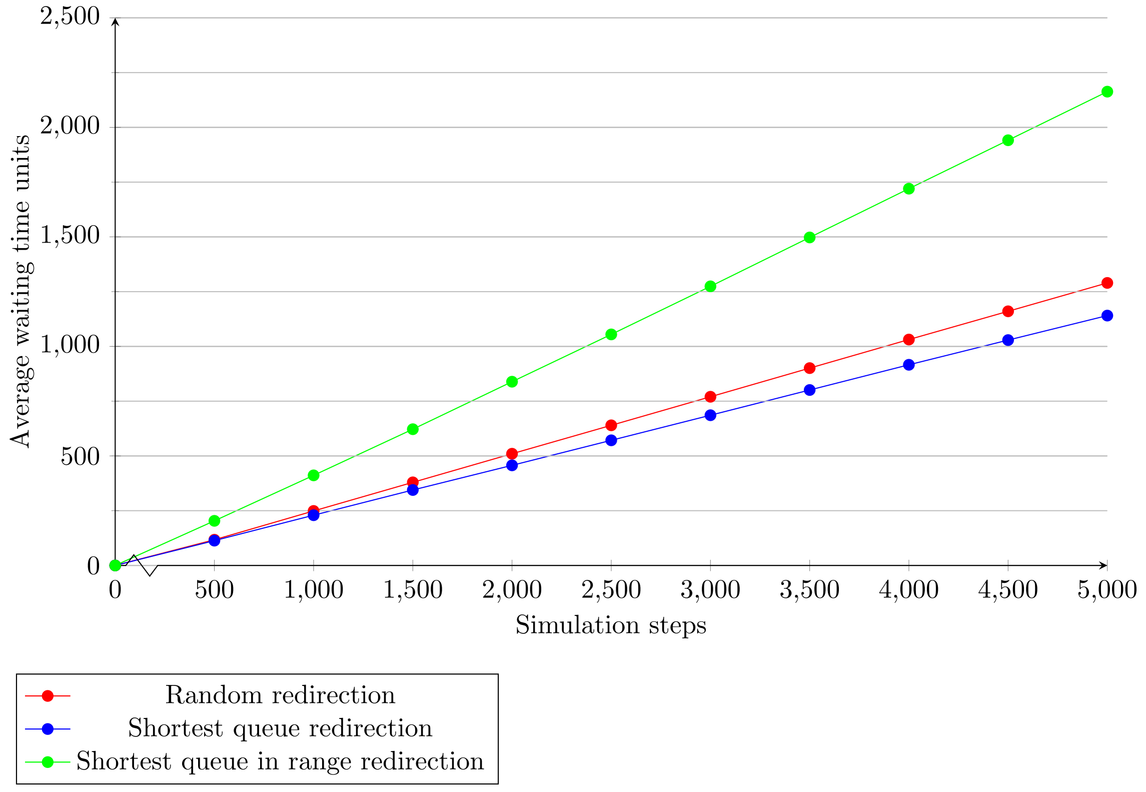

Figure 3: Average waiting times for the three different redirection policies.

As shown in the plot 3, the waiting times increase with the simulation steps. As expected, the waiting times for the *shortest queue redirection* are slightly better than the *random redirection* ones. Though, at first sight, the *shortest queue range redirection* seems to be the worst case, since its waiting time almost double the others. However, increasing waiting times also mean that visitors spend more time in queues and less time walking in order to reach attractions. Keeping in mind that the number of satisfaction is higher for this policy, this can be considered as a positive factor, since this result perfectly fits the policy's aim. So, the correct trend to evaluate is the waiting time per attraction, which is displayed in the plot 4.

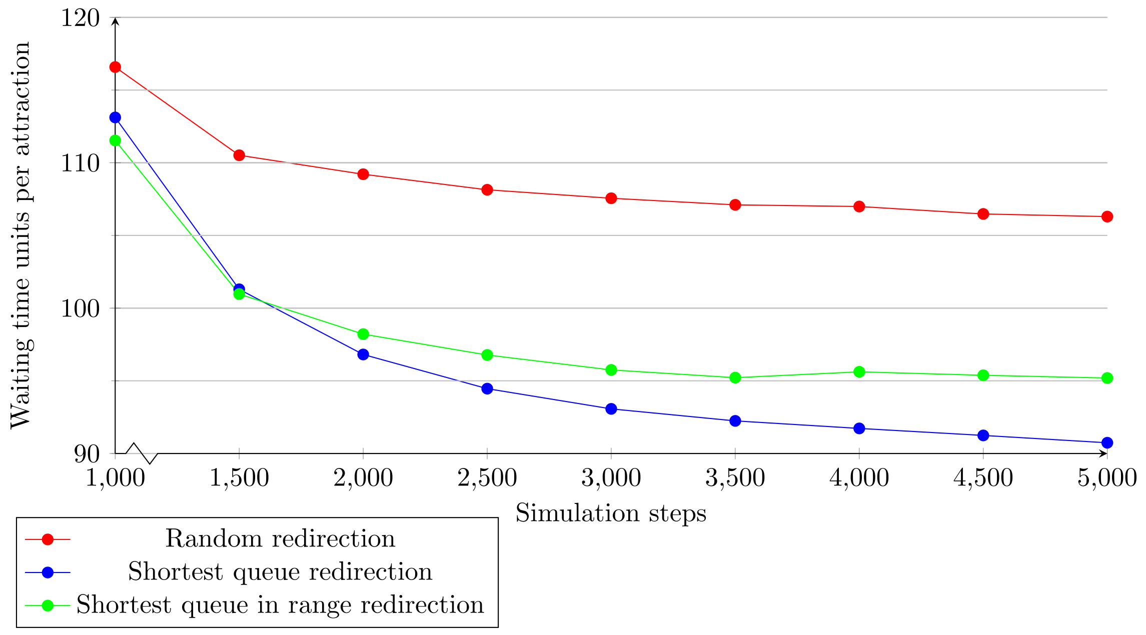

Figure 4: Waiting time per attraction for the three different redirection policies.

As shown in the plot, the *situated recommendation system* can reduce the average waiting time spent in queues. In particular, the *shortest queue redirection* performs slightly better than the *range* one. This happens because the *shortest queue range redirection* policy finds a **local optimum**: it only looks for attractions within a certain range, so it may not consider better options outside its range. Still, the latter grants the advantage of benefiting from a higher number of attractions.

5 Conclusions

This project was developed for the *Intelligent Cyber-Physical Systems* course. As already discussed in the introduction 1, sometimes it is not possible to implement complex systems due to uncertainty and limited resources. In the case of cyber-physical systems and infrastructures, this aspect is even more evident. Thus, the only way to demonstrate whether these systems can work or not is through proofs of concept. The implemented simulations can be considered valid proofs of concept as they provided sufficiently realistic information. This information can be used to predict scenarios and outcomes under determined conditions and to compare different strategies for recommendations.

One possible future development for the project is to switch language for the aggregate programming computations. In fact, *Alchemist* offers many *incarnations* that allow developers to build complex systems with different syntaxs and paradigms. For instance, a *work-in-progress* version of the simulation is being developed with the `scafi` incarnation. In fact, `scafi` is a *Domain Specific Language* for aggregate computing built on top of *Scala*. The advantages that it brings are a static type-checking system that makes it easier to develop complex simulations. Moreover, it allows performing aggregate and local computations, that belong to different paradigms, with the same language and data structures.

References

[1] D. Parisi. "Simulazioni. La realtà rifatta al computer". In: *Il Mulino* (2001). URL: https://www.mulino.it/isbn/9788815079039.

[2] D Pianini, S Montagna, and M Viroli. "Chemical-oriented simulation of computational systems with ALCHEMIST". In: *Journal of Simulation* 7.3 (Aug. 2013), pp. 202–215. DOI: 10.1057/jos.2012.27. URL: https://doi.org/10.1057%2Fjos.2012.27.

[3] Danilo Pianini, Mirko Viroli, and Jacob Beal. "Protelis: Practical aggregate programming". In: *Proceedings of the ACM Symposium on Applied Computing* (Jan. 2015), pp. 1846–1853.