Simulation of Large Scale Computational Ecosystems with Alchemist: A Tutorial

Danilo Pianini(✉)

Dipartimento di Informatica, Scienza e Ingegneria, Alma Mater Studiorum—Università di Bologna, 47522 Cesena, FC, Italy
danilo.pianini@unibo.it

Abstract. Many interesting systems in several disciplines can be modeled as networks of nodes that can store and exchange data: pervasive systems, edge computing scenarios, and even biological and bio-inspired systems. These systems feature inherent complexity, and often simulation is the preferred (and sometimes the only) way of investigating their behavior; this is true both in the design phase and in the verification and testing phase. In this tutorial paper, we provide a guide to the simulation of such systems by leveraging Alchemist, an existing research tool used in several works in the literature. We introduce its meta-model and its extensible architecture; we discuss reference examples of increasing complexity; and we finally show how to configure the tool to automatically execute multiple repetitions of simulations with different controlled variables, achieving reliable and reproducible results.

Keywords: Simulation · Pervasive computing · Self-organization

1 Introduction

The growing complexity of modern coordinated systems, fostered by trends such as cyber-physical systems (CPSs) and the Internet of Things, is a driving force behind novel development languages and methods. Techniques commonly used in classic centralized software development fall short when the system at hand features distribution, heterogeneity, and asynchronous communication. One relevant issue when developing software for distributed systems is testing: local execution of tests, the most common way of verifying system behavior before deployment, gets much harder when multiple distributed components are under test. Even worse, many such systems are (self-)adaptive, and feature autonomic behavior in face of unpredictable events, such as mobility, communication failures, or devices entering and leaving the system. In this context, simulation is key: by capturing the appropriate level of abstraction, simulators can provide insights into the coordinated system behavior without resorting to complex deployments. Moreover, they allow for experimenting with extreme situations, unlikely or even plain impossible, and explore the behavior of the system under test when subject to such extreme events.

Published by Springer Nature Switzerland AG 2021
M. Matos and F. Greve (Eds.): DAIS 2021, LNCS 12718, pp. 145–161, 2021.
https://doi.org/10.1007/978-3-030-78198-9_10

In this paper, we provide a tutorial introduction to Alchemist[1] [21], a simulation platform that has been successfully leveraged in the literature for a variety of different scenarios; including crowd tracking and steering, distributed algorithms evaluation, resource management, and even morphogenesis of multicellular organisms. The paper is organized as follows: Sect. 2 introduces the meta-model of the simulator and its extension system; Sect. 3 provides a sequence of increasingly richer examples, guiding the reader to an operative understanding of the simulator; Sect. 4 discusses how the simulator can be leveraged to extract data from multiple repetitions of simulations, and how specific executions can be reproduced exactly, thus allowing to debug and explore the system behavior in case rare anomalies are discovered; finally, Sect. 5 concludes the work.

2 A Meta-model of Computational Ecosystems

When designing a simulator, the most important decision to take is *what* the simulator should simulate, namely, what is its domain model. The decision is rather thorny, as decisions on the model are critical for optimization: the simpler the model, the larger the number of assumptions that can be made, the larger space for the designer to write specific optimization. The usual engineering process behind the creation of a simulator involves a selection of the model features, the decision of the time model (usually, the choice is between time-driven and event-driven), and finally the design of an engine that can support the execution in time of the model. Alchemist is atypical in this sense: since at the time of its conception performance was a major issue, it was conceived as an experimental tool meant to understand whether existing (and proven to be fast) engines could be extended to support the desired model without losing too much performance.

Alchemist is a by-product of the SAPERE European Project [33], where the goal was to coordinate pervasive software ecosystems with a biochemical and ecological metaphor. The idea behind the simulator was to extend existing high-performance stochastic chemistry simulation algorithms, adding support for:

- multiple interconnected "compartments" (containers of molecules),
- complex data types,
- non-exponentially distributed events,
- richer environment manipulation (chemical reactions are rather simple),
- different rules (reactions) in different compartments.

The latter requirement, in particular, restricted the choice of possible engine algorithms, ruling out the one identified by Slepoy et al. [25], thus orienting the choice towards the Gibson-Bruck algorithm [11]. Adaptation to the new model required important changes to the engine optimization structure, but it succeeded in preserving most of the performance [21]. Details on the internals of the simulator are out of the scope of this work and are documented in [21].

[1] https://alchemistsimulator.github.io.

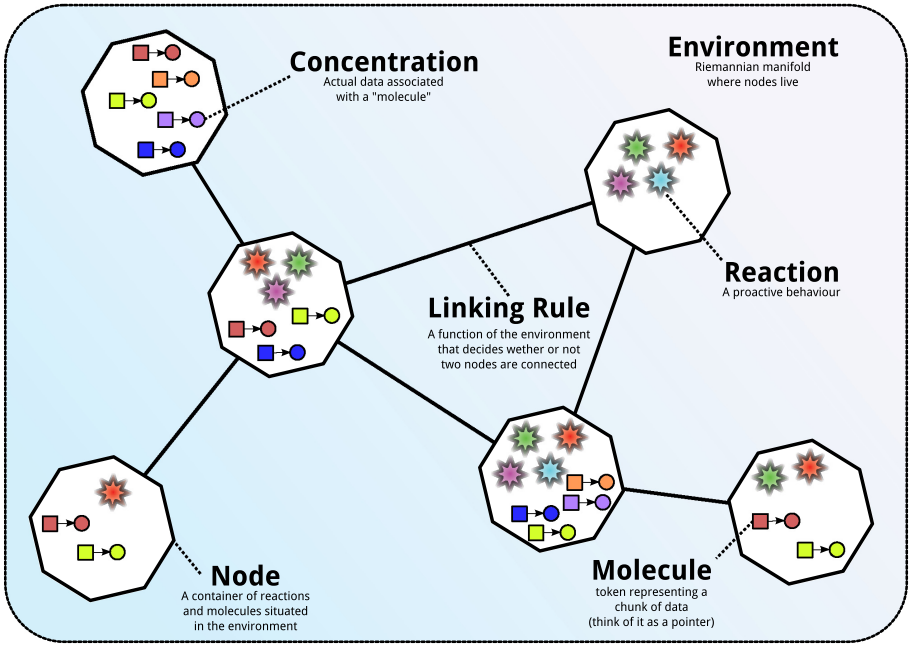

Fig. 1. A pictorial representation of the Alchemist simulator meta-model.

2.1 Abstract Meta-model

The abstract meta-model of Alchemist is hence influenced by its chemical origin. A pictorial view of such meta-model is presented in Fig. 1; the meta-model has the following entities:

- **Environment:** a Riemannian manifold modeling space, containing nodes and mapping them to positions;
- **Position:** a coordinate in the environment;
- **Node:** A container of molecules and reactions, living inside the environment;
- **Molecule:** the name of a data item;
- **Concentration:** the value associated with a particular molecule;
- **Linking rule:** a function associating each node to a neighborhood;
- **Neighborhood:** a structure composed of one central node and a set of other nodes considered its neighbors;
- **Reaction:** an event changing the status of the environment, made of a (possibly empty) set of conditions, a time distribution, a rate equation, and at least one action;
- **Condition:** a function of environment outputting a positive real number (usually but not necessarily interpreted as a boolean);
- **Action**: a change in the environment.

In this model, the sole source of changes in the environment is the execution of reactions, whose structure is depicted in Fig. 2. Alchemist is indeed a discrete

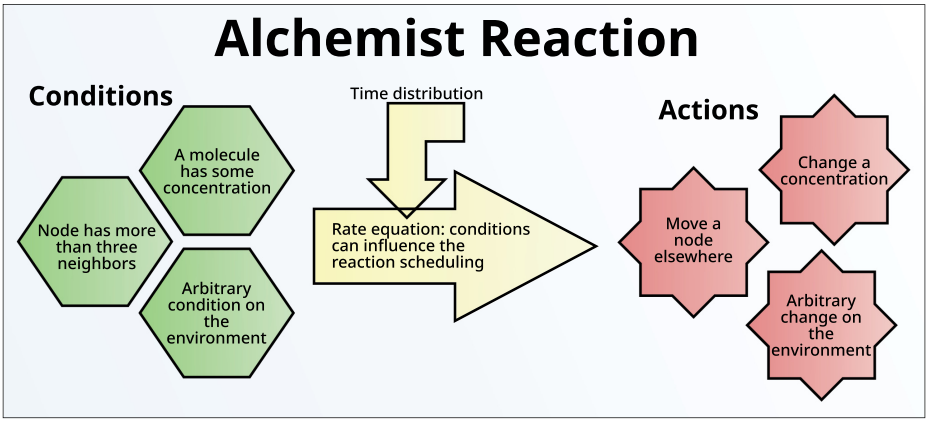

Fig. 2. A pictorial representation of the Alchemist simulator model of reaction.

event simulator, where reactions happen atomically. The peculiar structure with conditions and actions associated with nodes, along with a concept of context (which is omitted from the main discussion here, as it can be considered an implementation detail from the point of view of the user), allow Alchemist to build and maintain a dependency graph, speeding up significantly the scheduling of events [25].

2.2 Executable Models: Incarnations

In the previous discussion, we did not describe precisely what concentration is. Indeed, all Alchemist entities abstract from the type associated with the concentration; which has been purposely designed as abstract to allow extensibility. An Alchemist component fixing the concentration type is called an *incarnation*. Incarnations usually come not just with a well-defined type of concentration, but with complementary conditions, actions, reactions, and, possibly, means to create them from strings with custom syntax. Currently, the simulator features four incarnations: biochemistry, sapere, protelis, and scafi.

The *biochemistry* incarnation models concentration as a floating-point number. It provides a way to write specifications in a custom language derivative of chemical reactions, as well as custom nodes modeling the behavior of cells (for instance, supporting chemotaxis via polarization [7]). This incarnation has been used in the literature, for instance, to run stochastic simulations of the morphogenesis of Drosophila Melanogaster starting from a single cell [15].

The *sapere* incarnation is the oldest incarnation of the simulator and is meant to provide support for designing self-organizing mechanisms with the SAPERE [33] metaphor. In this incarnation, the concentration is a set of ground Linda-like tuples [10] matching a template (represented by a molecule); reactions are chemical-like rewriting rules, and nodes thus become programmable tuple spaces. This incarnation has been used in the literature for simulating, for

instance, crowd evacuation [16] and steering[2], anticipatory adaptation [14], and resource discovery [27].

In the *protelis* incarnation, concentration is defined as Java `Object`; in fact, this incarnation is meant to let nodes execute specifications written in the Protelis aggregate programming language [22], which can manipulate arbitrary data structures. This is likely the incarnation most commonly used in the literature, featuring dozens of examples including crowd tracking [3] and dispersal [30][3], target counting [20][4], drone coordination [5], and several distributed algorithms [1,2,29] and patterns [19].

Finally, the *scafi* incarnation provides support for the Scafi Scala DSL for aggregate programming [31]. Similar to Protelis, Scafi can manipulate arbitrary data types, hence the incarnation defines `Any` as concentration type. In the literature, Scafi has been exercised, for instance, in distributed peer-to-peer chats [5] and situated problem-solving [4].

3 Guided Examples

In this section, we will introduce a sequence of increasingly rich scenarios, showcasing the tool's capabilities and providing a path for learning the basics. The reader might experiment with the tool while reading this paper: the simulator comes with minimal requirements (a Java Virtual Machine 11 or newer), and a repository[5] [18] containing all examples and instructions is provided. In this paper, we will not go through all the technical details related to the environment setup (also, these details might change in the future with new versions of the tool); rather, we recommend checking out the simulator website and online manual[6]. All the code snippets presented here refer to the simulator at version 11.0.0.

3.1 The YAML-Based Simulation Specification Language

The extensible incarnation-based architecture of the simulator provides flexibility on the one hand, but it inevitably complicates the environment configuration, as it opens a potentially limitless range of options. On the one hand, the user must be able to write and use their custom extensions, but on the other hand, the simulation specifications should be as declarative, succinct, and human-readable as possible. Alchemist found a sweet spot between these requirements by relying on the YAML data serialization standard[7], a superset of JSON [6]. For the sake of self-containedness, we here introduce a few features of YAML which will be largely used in the remainder of the paper.

[2] A video is available at https://www.youtube.com/watch?v=QkWDynuELuo.
[3] A video is available at https://www.youtube.com/watch?v=606ObQwQuaE.
[4] A video is available at https://www.youtube.com/watch?v=MOwS6vQnubY.
[5] https://github.com/DanySK/DisCoTec-2021-Tutorial.
[6] https://alchemistsimulator.github.io/.
[7] https://yaml.org/spec/1.2/spec.html.

A YAML Primer. YAML scalars can be integers (e.g., `42`), floating-point numbers (e.g., `4.1` or `6.022e+23`), null values (`null`, `~`), booleans (`true`, `false`), or strings. Strings in YAML do not need to be quoted, although they can be if disambiguation is needed (for instance, `"42"` and `"42"` evaluate to strings rather than integers). Comments are prefixed by `#`. YAML mappings are key-value pairs (similar to dictionaries or hashes in other languages). They can be written by using a JSON-like syntax:

```
{one: 1, two: 2}
```

but they are usually expressed in a more human-readable form:

```
one: 1
two: 2
```

Mappings can be nested, and the two styles can be freely mixed. Nested mappings in human-readable form rely on semantic indentation to determine hierarchies:

```
integers:
  the_answer: 42
  the_beast: 666
floats: { avogadro: 6.022e+23, gravitational: 6.67408e-11 }
```

In case keys are repeated multiple times, the syntax is considered valid but only the latter entry is considered. YAML sequences are ordered collections of possibly equal elements, similar to arrays or lists in other languages. Similar to mappings, they feature two forms: a JSON-like syntax where elements are comma-separated and surrounded by square brackets, and a semantic-indented form:

```
- this is an element of the lists
- the following element is a nested list
- [one element of the nested list, another element of the nested list]
- the following elements are a nested list as well
  - one element of the nested list
  - another element of the nested list
```

Besides readability, one key reason to pick YAML over other formats is that it allows two forms of the "don't repeat yourself" (DRY) principle to be applied: reuse by anchoring, and reuse by merge keys:

```
some_map: &my_map #the content of some_map can now be referred to!
  { the_answer: 42, the_beast: 666 }
list_of_map_copies: [*my_map, *my_map, *my_map] # multiple copies
merged:
  the_trinity: 3
  <<: *my_map # map contents are merged into this map
  the_beast: absent # one value of the merged map is redefined
# Contents of merged: { the_trinity: 3, the_answer: 42, the_beast: absent }
```

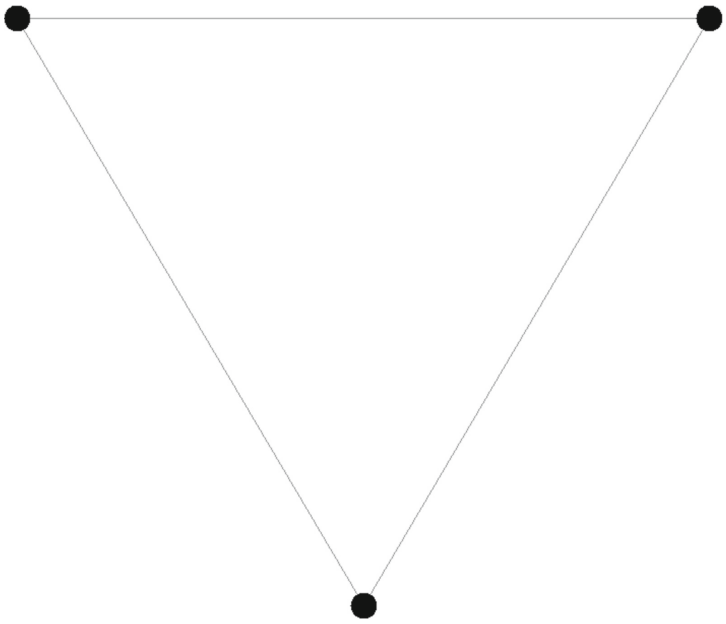

Fig. 3. Three devices in a bidimensional space.

3.2 Three Connected Devices

The Alchemist YAML specification must be a mapping containing at least the `incarnation` key, expecting a string value. Node locations are expressed in the `deployments` section while linking rules are specified in the `network-model` section. Crucially, Alchemist allows context-sensitive loading of arbitrary Java classes (or Java-compatible, thereby including Kotlin, Groovy, and Scala) implementing the required interfaces by specifying a map with a `type` key expecting an associated string value and (optionally) a `parameters` key expecting a sequence. As a first example, we write a specification displacing three nodes in a Euclidean bi-dimensional space, specifying a connection rule that makes all of them connected.

```
incarnation: sapere # The incarnation is always mandatory
network-model:
  type: ConnectWithinDistance # class name, must implement LinkingRule
  parameters: [2] # Comm. radius (argument of the class' constructor)
deployments:
  - type: Point # Loads a class with this name implementing Deployment
    parameters: [0, 0] # Coordinates
  - { type: Point, parameters: [0.5, 0.85] }
  - { type: Point, parameters: [-0.5, 0.85] }
```

The simulator, as depicted in Fig. 3 shows three connected devices occupying the specified positions. This simulation is not of great use, though: since there are no reactions, the simulation concludes as soon as it is started, with the simulation time going instantly to infinity.

3.3 A Grid of Devices Playing Dodgeball

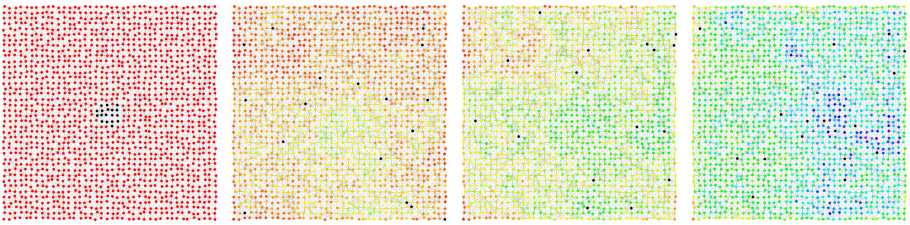

Fig. 4. Four subsequent snapshots of the simulation of the "dodgeball" example. Devices with a ball are depicted in black. All other devices' color hue depends on the hit count, shifting from red (zero hits) towards blue. (Color figure online)

We now showcase a more significant example, by deploying a grid of devices and making them play dodgeball. The program to be injected is rather simple: some nodes node will begin the simulation with a `ball`, and their goal will be to throw it to a random neighbor; whichever node gets hit takes a point, updates its score, and throws the ball again. This program is easy to write in a network of programmable tuple spaces, hence we write the following specification using the SAPERE incarnation.

```
incarnation: sapere
network-model: { type: ConnectWithinDistance, parameters: [0.5] }
deployments:
  type: Grid
  parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1] # A perturbed grid
  contents:
    - molecule: "{hit, 0}" # Everywhere, no one has been hit
    - in: { type: Rectangle, parameters: [-0.5, -0.5, 1, 1] } # In this area...
      molecule: ball # ...every node has a ball
  programs:
    - time-distribution: 1 # Rate of program evaluation
      # 'program' strings get interpreted by the incarnation
      program: "{ball} {hit, N} --> {hit, N + 1} {launching}" # Hit taken
    - program: "{launching} --> +{ball}" # Throw the ball to a neighbor ASAP
```

This short specification covers several features at once and deserves an explanation. For every entry in the `deployments` section, `contents` and `programs` can be specified describing respectively the initial content of nodes and their behavior. Every entry under `contents` contains a `molecule`, the corresponding `concentration`, and optionally one or more `shapes` defining where these contents should be inserted (no shape means "in every node"). Molecule and concentration descriptors must be interpreted by the selected incarnation; more in general, whenever there is no explicitly specified implementation of the concept to be used specified via `type`/`parameters` key, the provided data is sent to the incarnation. This is also true for `time-distribution` and `program` entries found in elements of the `programs` section. Snapshots for the running example are provided in Fig. 4.

3.4 A Gradient on a Grid of Devices

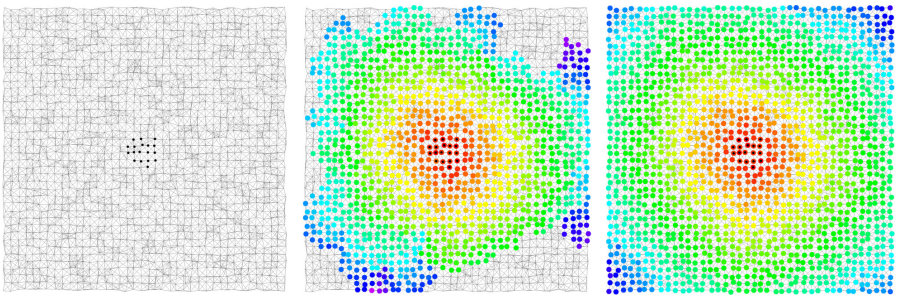

Fig. 5. Three subsequent snapshots of the simulation of the "gradient" example. Source devices have a central black dot. Devices' color hue depends on the gradient value, shifting from red (low) towards blue (high). (Color figure online)

Preparing more interesting computations is then a matter of writing more interesting specifications. In the following snippet, we implement a very simple specification of a gradient, a pattern that is considered to be the basis of many other patterns [8,9,29]. The specification is very similar to the previous one, and does not introduce any new concept:

```
incarnation: sapere
network-model: { type: ConnectWithinDistance, parameters: [0.5] }
deployments:
  type: Grid
  parameters: [-5, -5, 5, 5, 0.25, 0.25, 0.1, 0.1]
  contents:
    in: { type: Rectangle, parameters: [-0.5, -0.5, 1, 1] }
    molecule: source # Here is the source of the gradient
  programs:
    - time-distribution: 0.1 # Exponential with lambda=0.1
      # If there is a source, then the gradient is zero.
      program: "{source} --> {source} {gradient, 0}"
    - time-distribution: 1 # Exponential distribution with lambda=1
      # Send all neighbors your gradient value plus one
      program: "{gradient, N} --> {gradient, N} *{gradient, N+1}"
      # In case of multiple gradients, take the shortest
    - program: "{gradient, N}{gradient, def: N2>=N} --> {gradient, N}"
    - time-distribution: 0.1
      program: "{gradient, N} --> {gradient, N + 1}" # Aging process
    - program: "{gradient, def: N > 30} -->" # Death process
```

Snapshots of the execution can be found in Fig. 5.

3.5 Arbitrary Network Graphs

We now introduce more elaborate deployments. Right now, we experimented with points and grid, but the simulator offers much more flexibility. Besides deploying on points and grid, the simulator features:

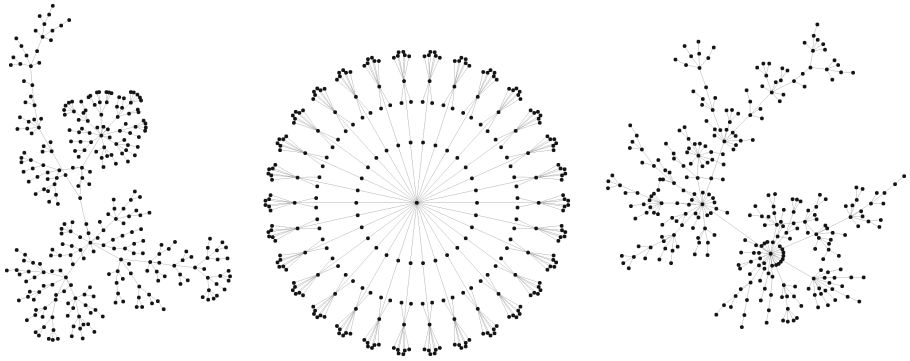

Fig. 6. A single environment with three advanced deployments. From left to right: a Lobster graph, a banana tree, and a scale-free network with preferential attachment.

- uniform random deployments within circles, eclipses, and polygons;
- deployments on (possibly perturbed) regular shapes (e.g., arcs);
- deployments in arbitrary positions;
- deployments relative to the current deployment (e.g., for adding nodes probabilistically close to others); and
- deployments on generated arbitrary graphs, via GraphStream [23].

To showcase some of the possibilities, we generate a deployment composed of three separate graphs: a Lobster graph [17], a Banana tree [28], and a scale-free network built with preferential attachment [12], using the snippet below.

```
incarnation: sapere
network-model: { type: ConnectWithinDistance, parameters: [0.5] }
deployments:
# parameters are node count, horizontal offset, vertical offset, zoom, graph type
  - { type: GraphStreamDeployment, parameters: [300, -30, 0, 0.8, Lobster, [5, 15]] }
  - { type: GraphStreamDeployment, parameters: [300, 0, 0, 2, BananaTree, 10] }
  - { type: GraphStreamDeployment, parameters: [300, 30, 0, PreferentialAttachment] }
```

The resulting environment is depicted in Fig. 6.

3.6 Node Mobility and Indoor Environments

All the previous examples run on a static network of devices in an endless bi-dimensional space. However, many interesting scenarios the simulator targets require mobility and a richer environment. In the following example, we show a group of mobile devices estimating the distance from a point of interest (the altar) while moving within a church, whose planimetry has been taken from an existing building. For the sake of simplicity, the movement here is modeled as a Lévy Walk [32], selected as it is a reasonable approximation of human walking [24]. Since self-stabilizing gradient on a mobile mesh network requires some tweaks to work appropriately [1], we switched to the Protelis incarnation and used the protelis-lang library [9] to implement the desired behavior in few lines of code.

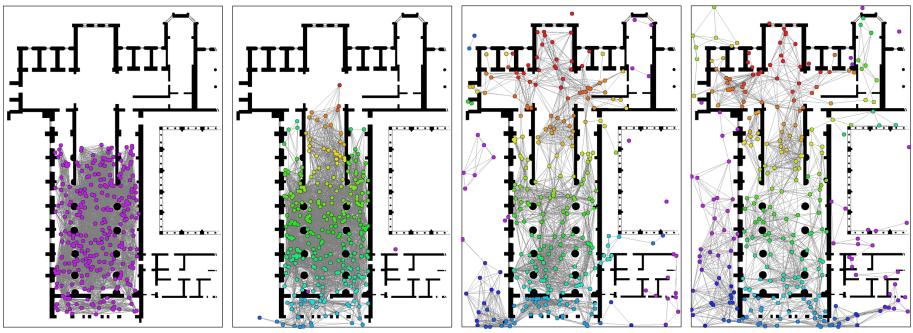

Fig. 7. Four snapshots of the simulation of mobile devices in a church. Devices progressively explore the location, while measuring the distance from a point of interest via gradient (red nodes are closer to the point of interest; purple ones are farther). (Color figure online)

```
incarnation: protelis
environment: { type: ImageEnvironment, parameters: [chiaravalle.png, 0.1] }
network-model: { type: ObstaclesBreakConnection, parameters: [50] }
deployments:
  type: Rectangle
  parameters: [300, 62, 15, 95, 200]
  programs:
    - time-distribution: 1
      program: >
        import protelis:coord:spreading
        let vector = self.getCoordinates() - [110, 325]
        distanceTo(hypot(vector.get(0), vector.get(1)) < 50)
    - program: send # Actual network message delivery
    - { type: Event, time-distribution: 1, actions: { type: LevyWalk, parameters: [1.4] } }
```

Comparing the code of this example with those in the previous sections highlights how the flexibility introduced by the incarnation mechanisms allows for programming scenarios with entirely different paradigms without changing the simulator. Besides the change of incarnation (and, hence, in the language used to program nodes), the most relevant change is the selection of a custom class for the environment, which loads an image turning pixels of the selected color into physical obstacles. Simulation snapshots are provided in Fig. 7.

3.7 Real-World Maps and GPS Traces

Finally, one relevant feature of Alchemist as a simulator for modern coordinated systems is the ability to exploit real-world geospatial data. The simulator can load data from OpenStreetMap exports, navigate devices towards a destination along streets by relying on GraphHopper[8] or by using GPS traces in GPX format, or even using the navigation system to interpolate sparse GPS traces, thus preventing nodes from taking impossible paths. Streets available for navigation

[8] https://www.graphhopper.com/.

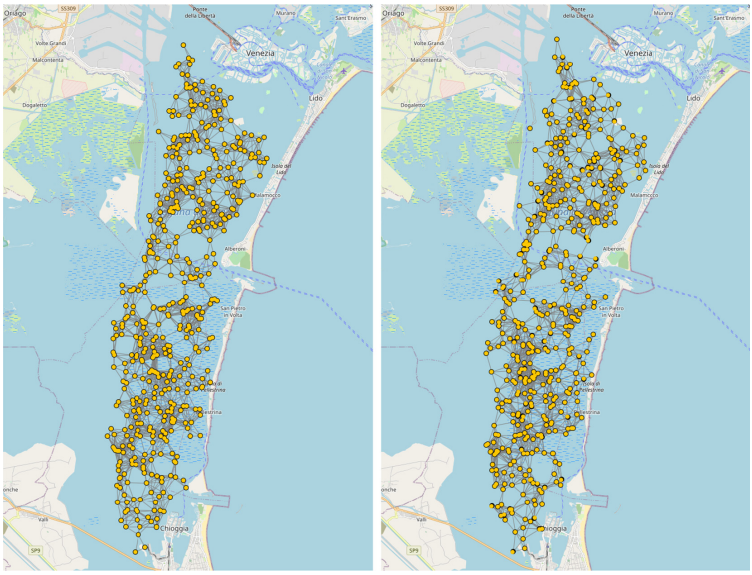

Fig. 8. 500 buoys deployed on the Venice lagoon.

can be selected based on the role of the node, which can be, for instance, bike, car, foot, wheelchair, hiking. For the sake of simplicity, we here present a simple scenario where water buoys are deployed in the Venice lagoon.

```
incarnation: sapere
environment: { type: OSMEnvironment }
network-model: { type: ConnectWithinDistance, parameters: [1000] }
_venice_lagoon: &lagoon
  [[45.20381, 12.25044], [45.22074, 12.26417], ..., [45.20381, 12.25044]]
deployments:
  type: Polygon
  parameters: [500, *lagoon]
  programs:
    time-distribution: 10
    type: Event
    actions: { type: BrownianMove, parameters: [0.0005] }
```

The complete sequence of coordinates is available as part of the Alchemist distribution[9] (all the examples presented in this work became part of the regression test suite of the simulator). Snapshots of the simulator are provided in Fig. 8.

4 In-Depth Analysis of Simulated Scenarios

Setting up a simulated environment is only the first step towards a scientific analysis. Usually, the system goes through a phase of debug and refinement and

[9] https://bit.ly/3cCfdnj.

is finally evaluated by analyzing its behavior by considering the metrics of choice under varying conditions.

Debugging a simulation requires the ability to reproduce the same behavior multiple times: an unexpected behavior requiring investigation may happen far into the simulation, or in corner conditions encountered by chance. Some simulators found in the literature, such as EdgeCloudSim [26], offer no possibility to exert such control. Alchemist was instead carefully crafted to guarantee reproducibility. Randomness is controlled by setting the random generator seeds separately for the deployments and the simulation execution, allowing for running different simulations on the same random deployment. Seeds are set at the top level of the simulation specification, as in the following snippet: `seeds: { simulation: 0, scenario: 0 }`. By default, Alchemist uses the Mersenne Twister pseudo-random generator, which guarantees high performance and very high dimensionality [13].

Alchemist provides first-class support for executing multiple simulations with varying conditions. Variables can be listed in the `variables` section of the simulation descriptor. Every variable has a default value and a way to generate other values. When a batch execution is requested, the cartesian product of all possible values for the selected variables is produced, the default values are used for non-selected variables, and then for each entry, a simulation is prepared and then executed (execution can be and usually is performed in parallel). Linear, logarithmic, and arbitrary variables are provided with the distribution, but custom generators can be easily implemented. Moreover, to favor reusability and apply the DRY principle, the simulator allows defining variables whose values possibly depend on values of other variables. Their values can be expressed in any JSR223[10]-compatible language (thereby including Scala, Kotlin, JavaScript, Ruby, Python, and Groovy), using Groovy as default. The variable definition mechanism is a powerful tool for creating rich simulations: values can be reused throughout the simulation specification, and can be computed based on other values, allowing for very advanced mechanisms such as environment sensitivity (e.g., react to the values of arbitrary environment variables), and dynamic simulation updates (e.g., by downloading components from a server).

Finally, the simulator provides tools for exporting data automatically. An `export` section on the simulation file instructs which data is considered interesting, and should be thus exported with the selected sampling frequency. Data can be exported separately for each node, or can be aggregated on the fly using any univariate statistic function (e.g., mean, sum, product, percentile, median...). The treatment of missing or non-finite values can be specified as well. Results are exported in comma-separated values files, easily importable in a variety of data analysis tools.

The following snippet showcases the aforementioned features by enriching the example presented in Sect. 3.6 with:

1. variables for the pedestrian walking speed, pedestrian count, and random seed;

[10] https://www.jcp.org/en/jsr/detail?id=223.

2. constants to ease the configuration of the simulation;
3. a Kotlin resource search expressed as a variable;
4. controlled reproducibility by controlling random seeds;
5. export of generated data (time and several statistics on the gradient).

```
variables:
  zoom: &zoom
    formula: 0.1 # Must be a valid Groovy snippet
  image_name: { formula: "'chiaravalle.png'" }
  image_path: &image_path
    language: kotlin # Pick whatever JSR223 language you like and add it to the classpath
    formula: > # The following is pure Kotlin code. other variables can be referenced!
      import java.io.File
      File("../..").walkTopDown().find { image_name in it.name }?.absolutePath ?: image_name
  # Linear free variable
  walking_speed: &walk-speed { default: 1.4, min: 1, max: 2, step: 0.1 }
  seed: &seed { default: 0, min: 0, max: 99, step: 1 } # 100 samples
  scenario_seed: &scenario_seed { formula: (seed + 31) * seed } # Variable-dependent
  people_count: &people_count
    type: GeometricVariable # A variable scanning a space with geometric segmentation
    parameters: [300, 50, 500, 9] # default 300, minimum 50, maximum 100, 9 samples
seeds: { simulation: *seed, scenario: *scenario_seed} # Controlled reproducibility
export: # One entry per column
  - time
  - molecule: "default_module:default_program"
    aggregators: [mean, max, min, variance, median] # From Apache's UnivariateStatistic
    value-filter: onlyfinite # discards NaN and Infinity
environment: { type: ImageEnvironment, parameters: [*image_path, *zoom] }
... # See the previous example for missing code
deployments:
  type: Rectangle
  parameters: [*people_count, 62, 15, 95, 200]
  programs:
    ... # See the previous example for missing code
    - {type: Event,time-distribution: 1,actions: {type: LevyWalk,parameters: [*walk-speed]}}
```

Its execution in normal mode picks the default values of every variable, producing a CSV file that includes detail on the variables values, information on the time at which the simulation began, and descriptors for the meaning of each data column. Execution in batch mode with a selection of free variables generates instead one file for each of the possible variables combination. The following is an excerpt of the file content:

```
#####################################################################
# Alchemist log file - simulation started at: 2021-03-30T14:11+0000 #
#####################################################################
# walking_speed = 1.4, seed = 0.0, people_count = 300.0
#
# The columns have the following meaning:
# time default_module:default_program[Mean] default_module:default_program[Max] default_m...
0.0 NaN NaN NaN NaN NaN
100.0011337332612 158.7760995571516 259.2936556081146 0.0 1973.392001590659 154.960773862537
200.0011337332612 191.2437429491354 387.346209438406 0.0 3633.63195819356 191.956119047278
```

5 Concluding Remarks

The role of simulation in the development of cutting-edge, coordinated, and situated systems is paramount: it provides a way to exercise complex setups and

corner situations from a single device, as well as ways to perform analysis to investigate the behavior of novel approaches and algorithms.

In this tutorial paper, we illustrated one simulation tool leveraged in several works in the literature: the Alchemist simulator. We first described its meta-model and explained the concept of "incarnation", opening the door to the simulation of a large variety of scenarios. Then, we dived into a sequence of increasingly complex examples, showcasing several of the many possibilities offered by Alchemist, among which: support for simulations in indoor environments, real-world maps, complex network graphs, and execution of aggregate programs [3]. Finally, we discussed how this tool can be leveraged for debugging the evolution of a complex networked system (by enforcing reproducibility), as well as for generating data to be fed to data analysis tools.

Acknowledgements. This work has been supported by the MIUR PRIN Project N. 2017KRC7KT "Fluidware".

References

1. Audrito, G., Damiani, F., Viroli, M.: Optimal single-path information propagation in gradient-based algorithms. Sci. Comput. Program. **166**, 146–166 (2018). https://doi.org/10.1016/j.scico.2018.06.002
2. Audrito, G., Pianini, D., Damiani, F., Viroli, M.: Aggregate centrality measures for IoT-based coordination. Sci. Comput. Program. **203**, 102584 (2021). https://doi.org/10.1016/j.scico.2020.102584
3. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the internet of things. IEEE Comput. **48**(9), 22–30 (2015). https://doi.org/10.1109/MC.2015.261
4. Casadei, R., Tsigkanos, C., Viroli, M., Dustdar, S.: Engineering resilient collaborative edge-enabled IoT. In: 2019 IEEE International Conference on Services Computing (SCC). IEEE (2019). https://doi.org/10.1109/scc.2019.00019
5. Casadei, R., Viroli, M., Audrito, G., Pianini, D., Damiani, F.: Engineering collective intelligence at the edge with aggregate processes. Eng. Appl. Artifi. Intelli. **97** (2021). https://doi.org/10.1016/j.engappai.2020.104081. http://www.sciencedirect.com/science/article/pii/S0952197620303389
6. David, V.: JSON: Main Principals. CreateSpace Independent Publishing Platform, North Charleston (2016)
7. Devreotes, P., Janetopoulos, C.: Eukaryotic chemotaxis: distinctions between directional sensing and polarization. J. Biol. Chem. **278**(23), 20445–20448 (2003). https://doi.org/10.1074/jbc.r300010200
8. Fernandez-Marquez, J.L., Serugendo, G.D.M., Montagna, S., Viroli, M., Arcos, J.L.: Description and composition of bio-inspired design patterns: a complete overview. Nat. Comput. **12**(1), 43–67 (2013). https://doi.org/10.1007/s11047-012-9324-y
9. Francia, M., Pianini, D., Beal, J., Viroli, M.: Towards a foundational API for resilient distributed systems design. In: 2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, 18–22 September 2017, pp. 27–32 (2017). https://doi.org/10.1109/FAS-W.2017.116

10. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985). https://doi.org/10.1145/2363.2433
11. Gibson, M.A., Bruck, J.: Efficient exact stochastic simulation of chemical systems with many species and many channels. J. Phys. Chem. A **104**(9), 1876–1889 (2000). https://doi.org/10.1021/jp993732q
12. Hidalgo, C.A., Barabási, A.: Scale-free networks. Scholarpedia **3**(1), 1716 (2008). https://doi.org/10.4249/scholarpedia.1716
13. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998). https://doi.org/10.1145/272991.272995
14. Montagna, S., Pianini, D., Viroli, M.: Gradient-based self-organisation patterns of anticipative adaptation. In: Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2012, Lyon, France, 10–14 September 2012, pp. 169–174 (2012). https://doi.org/10.1109/SASO.2012.25
15. Montagna, S., Pianini, D., Viroli, M.: A model for drosophila melanogaster development from a single cell to stripe pattern formation. In: Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, 26–30 March 2012, pp. 1406–1412 (2012). https://doi.org/10.1145/2245276.2231999. http://doi.acm.org/10.1145/2245276.2231999
16. Montagna, S., Viroli, M., Risoldi, M., Pianini, D., Di Marzo Serugendo, G.: Self-organising pervasive ecosystems: a crowd evacuation example. In: Troubitsyna, E.A. (ed.) SERENE 2011. LNCS, vol. 6968, pp. 115–129. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24124-6_12
17. Morgan, D.: All lobsters with perfect matchings are graceful. Electron. Notes Discret. Math. **11**, 503–508 (2002). https://doi.org/10.1016/S1571-0653(04)00095-2
18. Pianini, D.: Danysk/discotec-2021-tutorial: 0.1.0 (2021). https://doi.org/10.5281/ZENODO.4701062. https://zenodo.org/record/4701062
19. Pianini, D., Casadei, R., Viroli, M., Natali, A.: Partitioned integration and coordination via the self-organising coordination regions pattern. Future Gener. Comput. Syste. **114**, 44–68 (2021). https://doi.org/10.1016/j.future.2020.07.032. http://www.sciencedirect.com/science/article/pii/S0167739X20304775
20. Pianini, D., Dobson, S., Viroli, M.: Self-stabilising target counting in wireless sensor networks using Euler integration. In: 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA, 18–22 September 2017, pp. 11–20 (2017). https://doi.org/10.1109/SASO.2017.10
21. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simul. **7**(3), 202–215 (2013). https://doi.org/10.1057/jos.2012.27
22. Pianini, D., Viroli, M., Beal, J.: Protelis: practical aggregate programming. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, 13–17 April 2015, pp. 1846–1853 (2015). https://doi.org/10.1145/2695664.2695913. http://doi.acm.org/10.1145/2695664.2695913
23. Pigné, Y., Dutot, A., Guinand, F., Olivier, D.: Graphstream: a tool for bridging the gap between complex systems and dynamic graphs. CoRR abs/0803.2093 (2008). http://arxiv.org/abs/0803.2093
24. Rhee, I., Shin, M., Hong, S., Lee, K., Kim, S.J., Chong, S.: On the levy-walk nature of human mobility. IEEE/ACM Trans. Netw. **19**(3), 630–643 (2011). https://doi.org/10.1109/tnet.2011.2120618
25. Slepoy, A., Thompson, A.P., Plimpton, S.J.: A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks. J. Chem. Phys. **128**(20), 05B618 (2008). https://doi.org/10.1063/1.2919546

26. Sonmez, C., Ozgovde, A., Ersoy, C.: EdgeCloudSim: an environment for performance evaluation of edge computing systems. Trans. Emerg. Telecommun. Technol. **29**(11), e3493 (2018). https://doi.org/10.1002/ett.3493
27. Stevenson, G., Ye, J., Dobson, S., Pianini, D., Montagna, S., Viroli, M.: Combining self-organisation, context-awareness and semantic reasoning: the case of resource discovery in opportunistic networks. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC 2013, Coimbra, Portugal, 18–22 March 2013, pp. 1369–1376 (2013). https://doi.org/10.1145/2480362.2480619. http://doi.acm.org/10.1145/2480362.2480619
28. Swaminathan, V., Jeyanthi, P.: Super edge-magic strength of fire crackers, banana trees and unicyclic graphs. Discret. Math. **306**(14), 1624–1636 (2006). https://doi.org/10.1016/j.disc.2005.06.038
29. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul. **28**(2), 1–28 (2018). https://doi.org/10.1145/3177774
30. Viroli, M., Bucchiarone, A., Pianini, D., Beal, J.: Combining self-organisation and autonomic computing in CASs with aggregate-MAPE. In: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W). IEEE (2016). https://doi.org/10.1109/fas-w.2016.49
31. Viroli, M., Casadei, R., Pianini, D.: Simulating large-scale aggregate mass with alchemist and scala. In: Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, 11–14 September 2016, pp. 1495–1504 (2016). https://doi.org/10.15439/2016F407
32. Zaburdaev, V., Denisov, S., Klafter, J.: Lévy walks. Rev. Mod. Phys. **87**(2), 483–530 (2015). https://doi.org/10.1103/revmodphys.87.483
33. Zambonelli, F., et al.: Developing pervasive multi-agent systems with nature-inspired coordination. Pervasive Mobile Comput. **17**, 236–252 (2015). https://doi.org/10.1016/j.pmcj.2014.12.002