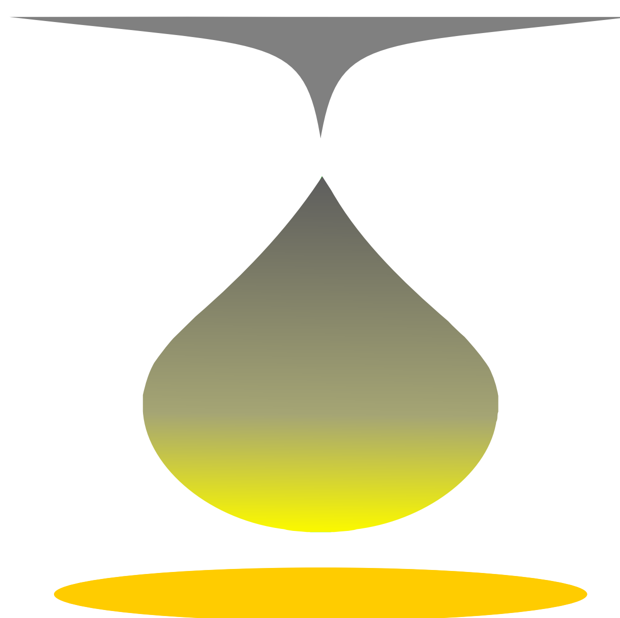
Università di Bologna

The Alchemist simulator

Full Manual



Contributors:
Michele Bombardi
Chiara Casalboni
Davide Ensini
Sara Montagna
Luca Nenni
Michele Pratiffi
Mirko Viroli

Author:
Danilo Pianini

Date:
November 2, 2013

Contents

1 Generalities **3**
1.1 introduction . . . 3
1.2 Computational Model . . . 4
1.3 Engine . . . 6
1.3.1 Dynamic Indexed Priority Queue . . . 7
1.3.2 Dynamic Dependency Graph . . . 8
1.4 Incarnations and language chain . . . 9

2 Using Alchemist **12**
2.1 Getting started . . . 12
2.2 Using the simulator via Java . . . 13
2.3 Chemistry . . . 13
2.4 ListDouble . . . 14
2.5 SAPERE . . . 14
2.5.1 Specific language . . . 14
2.5.2 Writing agents . . . 26
2.5.3 Built-in fast gradient for SAPERE . . . 27
2.5.4 Gradient and Force-based model for Pedestrian Dynamics . . . 28
2.6 Additional features . . . 37
2.6.1 Alchemist2Blender: An Alchemist to Blender interface 37
2.6.2 From png image to Environment . . . 39
2.6.3 Real world maps . . . 44
2.6.4 Approximate Probabilistic Model Checker . . . 46

3 How to develop Alchemist **57**
3.1 Mercurial . . . 57
3.2 Maven . . . 58
3.3 PMD . . . 58
3.4 Find Bugs . . . 58
3.5 Code style . . . 59

3.6 Final remarks for the devels 59
3.6.1 Use internal logger 59
3.6.2 Test plan 61
3.7 How to report issues 61

Chapter 1

Generalities

1.1 introduction

Complex systems, *i.e.* systems composed by many different parts that interact so as to generate an unpredictable emergent behaviour, are everywhere in nature – chemical, biological, social systems – and also in artificial systems realised to support nowadays scenarios, like pervasive computing systems built over the increasingly available smart computational devices.

Both natural and artificial systems of this kind have a set of common key properties:

- situatedness – they deal with spatially- and possibly socially-situated activities of entities, and should therefore be able to interact with the surrounding world and adapt their behaviour accordingly;

- adaptivity – they should inherently exhibit properties of autonomous adaptation and management to survive contingencies without external intervention, global supervision, or both;

- self-organisation – spatial and temporal patterns of behaviour should emerge out of local interactions and without a central authority that imposes pre-defined plans.

In order to find common models capturing these properties in a uniform and coherent way, a typical approach is to take inspiration from nature, adopting some of its metaphors to design innovating computing models [26]. In fact, in natural systems all the activities of the system components are inherently situated in space and driven by local interactions only. Such interactions are not ruled by pre-defined orchestrated patterns, but are rather simply subject

to a limited set of natural laws from which even complex patterns of interactions dynamically emerge via self-organisation. In this way, adaptivity becomes an inherent characteristic deriving from the existence of self-organising interactions patterns, whose structure can flexibly yet robustly re-shape in response to contingencies. In line with recent research in the field of pervasive computing [22, 24], among the many nature-inspired metaphors we choose a chemical one properly enriched with a concept of networked space and diffusion of "chemicals" through system locations—what we can refer to as a biochemical metaphor. This metaphor can be turned into a computational settings by adopting a reference meta-model that, developing on the ABM (agent-based model)[18], grounds around autonomous and possibly heterogeneous agents situated in an environment, and whose behaviour (internal and collaborative) is described in terms of chemical-like rules evolving (matching, diffusing) structured annotations that play the role of molecules and are created/manipulated/consumed by agents. A key brick of the engineering of such agent systems, apart from middleware and programming languages [23], is simulation, which is a necessary "tool" to analyse system behaviour prior to deployment, which is rather mandatory with systems required to expose emergent behaviour. Alchemist is meant to face natively the above model abstractions. It implements an optimised version of the Gillespie's SSA – which has in principle been developed to model the dynamic of chemical solutions – namely the Next Reaction Method [12], that is known for its high performances. To face the model requirements, this algorithm has been properly extended with the possibility to have dynamic reactions, *i.e.*, system transitions that can be added or removed during simulation due to network mobility and unpredicted situations.

The meta-model and simulation framework we are presenting here are pretty generic, and as such they can have a wide range of applications beside pervasive computing, including computational biology and social interactions.

1.2 Computational Model

A pictorial representation of the underlying computational model is shown in Figure 1.1. In this simple vision of the world, an environment is a multi dimensional space, continuous or discrete, which is able to contain nodes and which is responsible of linking them following a rule. The environment may or not allow nodes to move. Nodes are entities which can be programmed with a set of reactions possibly changing over time. They also contain molecules, each one equipped with a concentration value.

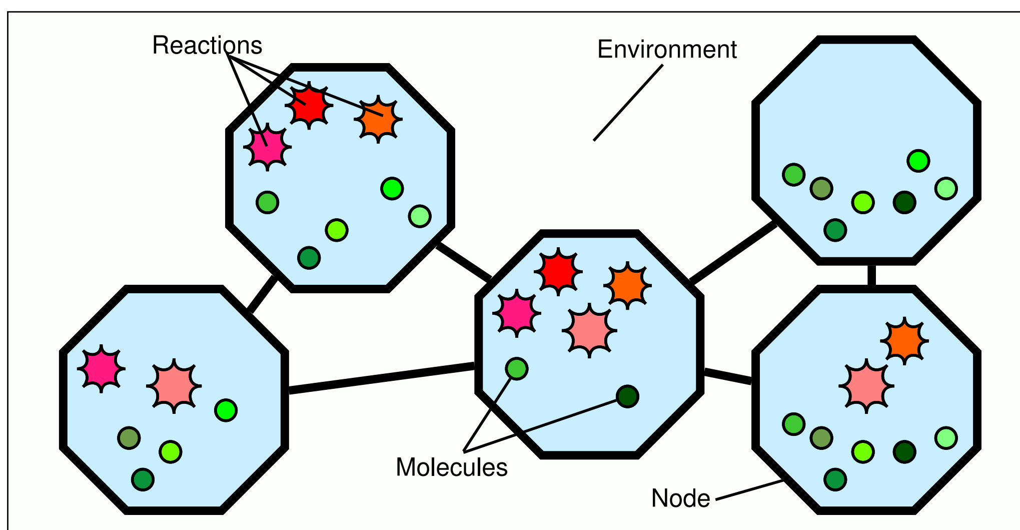


Figure 1.1: ALCHEMIST computational model: it features a possibly continuous space embedding a linking rule and containing nodes. Each node is programmed with a set of reactions and contains a set of structured molecules.

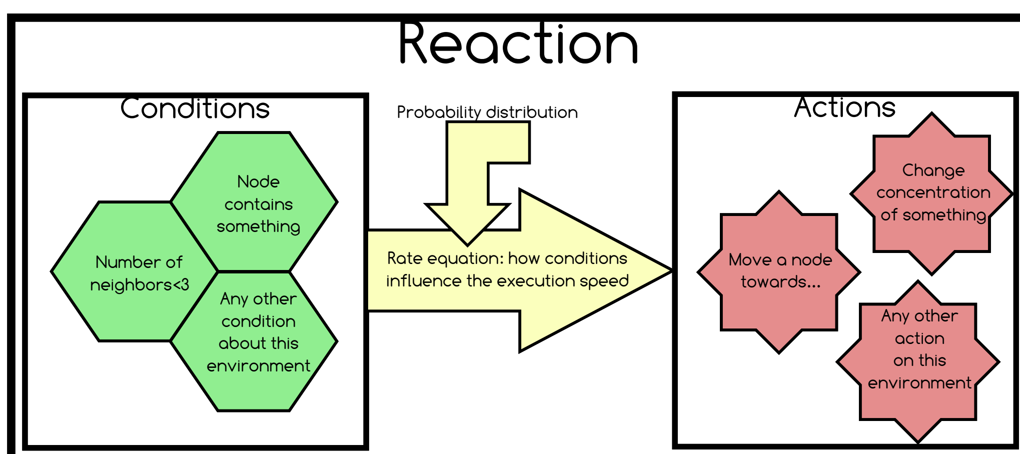


Figure 1.2: ALCHEMIST model of reaction: a set of conditions on the environment that determines whether the reaction is executable, a rate equation describing how the speed of the reaction changes in response to environment modifications, a probability distribution for the event and a set of actions, which will be the neat effect of the execution of the reaction.

The concept of reaction is more elaborated than that used in chemistry: in classical chemical models, a reaction lists a number of reactant molecules which, combined, produce a set of product molecules. This kind of description is too strict for a general purpose simulator. A more generic concept is to consider a reaction as a set of conditions about the system state, which triggers the execution of a set of actions. A condition is a function associating a boolean value to the current state of the system (or a subpart of it), an action is a procedure that modifies the annotations representing it. This allows, for example, to model reactions which are faster if a node has many neighbours, or also reactions that resemble complex biological phenomena such as the diffusion of morphogenes during embryo development as described in [17]. It also allows to define which kind of time distribution to use to trigger reactions: this enables us to model and simulate systems based on Continuous Time Markov Chains (CTMCs), to add triggers, or also to rely just on classical discrete time "ticks".

1.3 Engine

One of the most flexible and fast algorithms to model chemical systems like the above one is the Next Reaction Method presented in [12]. Alchemist follows the basic steps of this algorithm, as listed in Algorithm 1.

Algorithm 1 Simulation flow in Alchemist

```
 1: cur_time = 0
 2: cur_step = 0
 3: for each node n in environment do
 4:     for each reaction nr in n do
 5:         generate a new putative time for nr
 6:         insert nr in DIPQ
 7:         generate dependencies for nr
 8: while cur_time < max_time and cur_step < max_step do
 9:     r = the next reaction to execute
10:     if r's conditions are verified then
11:         execute all the actions of r
12:         for each reaction rd which depends on r do
13:             update the putative execution time
14:     generate a new putative time for r
```

Likewise other simulation approaches for chemistry [13, 21], Gibson's algorithm is not appropriate as-is to support our simulations: in particular, it does not provide facilities to add/remove/move nodes (which ultimately changes the set of reactions and their interdependencies dynamically), and to inject triggers and other non-exponential time distributed events. Hence, our work on the engine had the primary goal to extend the Next Reaction

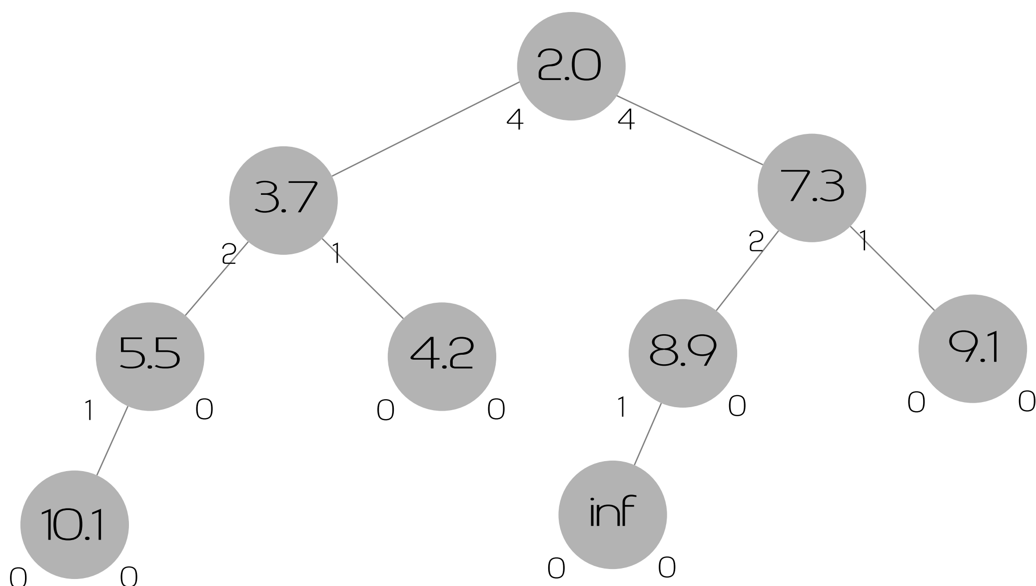


Figure 1.3: Indexed Priority Queue extended with children count per branch

Method providing the possibility to add and remove reactions dynamically. In order to add this support, it is mandatory to provide methods to add and remove reactions from two key data structures used in Next Reaction, namely, the indexed priority queue and the dependency graph, which are described in turn.

1.3.1 Dynamic Indexed Priority Queue

The Indexed Priority Queue (IPQ) is a data structure proposed in [12]. It is a binary tree of reactions, whose main property is that each node stores a reaction whose putative time of occurrence is lower than each of its sons. This means that the next reaction to execute is always in the root of the tree and can be accessed in constant time. A key property of the original IPQ is that the swap procedure used to update the data structure does not change the balance of the tree, ensuring optimal update times in every situation. This feature was easily achieved because no nodes were ever added neither removed from the structure. As a consequence, once the tree is balanced at creation time no event can occur to change its topology. This is no longer the case in Alchemist, and we have to provide an extension to the IPQ machinery in order to properly handle tree balancing.

The current solution is, for each node of the tree, to keep track of the number of children per branch, having in such way the possibility to keep the

tree balanced when adding nodes. In figure 1.3 we show how the same IPQ drawn in [12] would appear with our extension. Given this data structure, the procedures to add and remove a new node `n` are described respectively in Section 1.3.1 and Section 1.3.1, in which the procedure `UPDATE_AUX(n)` is the same described in [12].

Algorithm 2 Procedure to add a new node `n`

```
 1: if root does not exist then
 2:     n is the new root
 3: else
 4:     c ← root
 5:     while c has two children do
 6:         if c.right < c.left then
 7:             dir ← right
 8:         else
 9:             dir ← left
10:         add 1 to count of dir children
11:         c ← c.dir
12:         if c has not the left child then
13:             n becomes left child of c
14:             set count of left nodes of c to 1
15:         else
16:             n becomes right child of c
17:             set count of right nodes of c to 1
18:     UPDATE_AUX(n)
```

Algorithm 3 Procedure to remove a node `n`

```
 1: c ← root
 2: while c is not a leaf do
 3:     if c.left > c.right then
 4:         dir ← left
 5:     else
 6:         dir ← right
 7:     subtract 1 to count of dir children
 8:     c ← c.dir
 9: if c ≠ n then
10:     swap n and c
11:     remove n
12:     UPDATE_AUX(c)
13: else
14:     remove n
```

Using Section 1.3.1 and Section 1.3.1, the topology of the whole tree is constrained to remain balanced despite the dynamic addition and removal of reactions.

1.3.2 Dynamic Dependency Graph

Since we want to support natively and efficiently the interaction between nodes, which become dependencies among the reactions occurring in such

nodes, we defined three contexts (also called scopes): `local`, `neighborhood` and `global`. Each reaction has an input context and an output context dynamically computed, which respectively represent where data influencing the rate calculus is located and where the modifications are made.

The first issue to address is to evaluate if a reaction `r1` may influence another reaction `r2`, considering their contexts. We introduced a boolean procedure `mayInfluence(r1, r2)` which operates on two reactions and returns a true value if:

- `r1` and `r2` are on the same node OR
- `r1`'s output context is `global` OR
- `r2`'s input context is `global` OR
- `r1`'s output context is `neighborhood` and `r2`'s node is in `r1`'s node neighbourhood OR
- `r2`'s input context is `neighborhood` and `r1`'s node is in `r2`'s node neighbourhood OR
- `r1`'s output context and `r2`'s input context are both `neighborhood` and the neighbourhoods of their nodes have at least one common node.

Given this handy function, we can assert that a dependency exists between the execution of a reaction `r1` and another reaction `r2` if `mayInfluence(r1,r2)` is true and at least a molecule whose concentration is modified by `r1` is among those influencing `r2`.

Adding a new reaction implies to verify its dependencies against every reaction of the system. In case there is a dependency, it must be added to the dependency graph. Removing a reaction `r` requires to delete all dependencies in which `r` is involved both as influencing and influenced. Moreover, in case of a change of system topology, a dependency check among reactions belonging to nodes with modified neighbourhood is needed. It can be performed by scanning them, calculating the dependencies with the reactions belonging to new neighbours and deleting those with nodes which are no longer in the neighbourhood.

1.4 Incarnations and language chain

It is important to note that there is no restriction about the kind of data structure representing the concentration, which can in fact be used to model

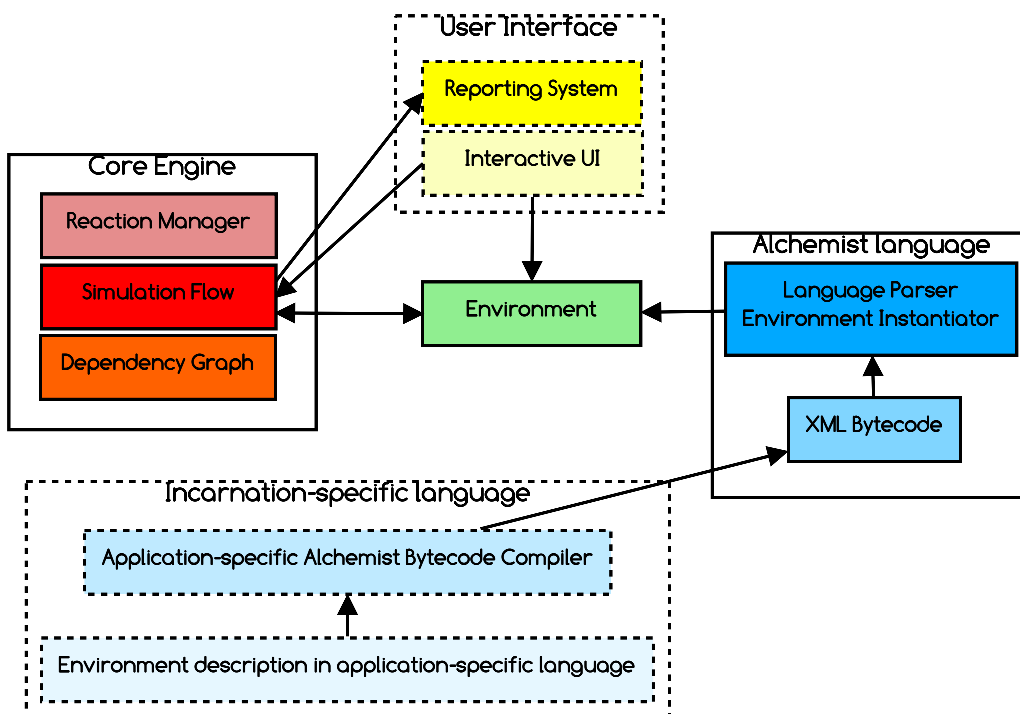


Figure 1.4: ALCHEMIST architecture. Elements drawn with continuous lines indicates components common for every scenario and already developed, those with dotted lines are extension-specific components which have to be developed with a specific incarnation in mind.

structured information: by defining a new kind of structure for the concentration, it is possible to incarnate the simulator for different specific uses. For example, by assessing that the concentration is an integer number, representing the number of molecules currently present in a node, ALCHEMIST becomes a stochastic simulator for chemistry. A more complex example can be the definition of concentration as a tuple set, and the definition of molecule as tuple template. If we adopt this vision, ALCHEMIST can be a simulator for a network of tuple spaces. Each time a new definition of concentration and molecule is made, a new "incarnation" of ALCHEMIST is automatically defined. For each incarnation, a set of specific actions, conditions, reactions and nodes can be defined, and all the entities already defined for a more generic concentration type can be reused.

Chapter 2

Using Alchemist

Alchemist is split in many different sub-projects. Some of them are generic and contain core components or utilities, other define new incarnations, and other (plugins) use those incarnations to realise specific scenarios. Documentation for every module of Alchemist is available in the official maven site (`http://alchemist-maven.apice.unibo.it/`).

Due to its modular nature, writing a simulation may differ between different incarnations. In this chapter we will see how to get Alchemist, and then we will explore some code examples for each of the available incarnations, both using specific languages and Java.

2.1 Getting started

After several months of development, Alchemist got its own standalone distribution. The latest version is available in the official project download page: `https://bitbucket.org/danysk/alchemist/downloads`. The standalone distribution allows to load and run the Alchemist XML files, using all the built-in abstractions. If the user has a set of additional parts (e.g. new actions, reactions, environments or conditions) he needs, it is possible to load those at runtime, by simply specifying a JAR file where all those facilities are located.

To develop new parts for the simulator, the user must download the source code from the main repository and compile it herself. The fastest way is by far using Eclipse. In order to help getting started, we realised a video which will guide the user throughout the process, it is available at `http://www.youtube.com/watch?v=C1nejv_9Wd4`.

To launch a graphical instance of Alchemist, the user can launch the class Alchemist, which can be found in the reporting module. Launching it

within Eclipse is warmly recommended, since it will take care of including in the classpath all the required libraries.

2.2 Using the simulator via Java

Using ALCHEMIST in via Java is pretty tricky and generally discouraged, however not every incarnation has its own specific language yet. Moreover, it may turn useful for those interested in building some stand alone application which uses the ALCHEMIST engine internally. The general work flow should be the one framed in Algorithm 4.

Algorithm 4 Generic procedure to initialize a simulation using Java

```
 1: create the environment
 2: for each node you want to add do
 3:     create the node
 4:     add the molecules you want in the node
 5:     for each reaction you want to add do
 6:         create the reaction
 7:         create a list of conditions
 8:         create a list of actions
 9:         for each condition do
10:             create the condition
11:             add the condition to the list
12:         for each action do
13:             create the action
14:             add the action to the list
15:         set the conditions list as reaction's conditions
16:         set the actions list as reaction's actions
17:     create a position for the node
18:     add the node to the environment
19: create the simulation
20: for each output monitor you want do
21:     create the corresponding object
22:     add the output monitor to the simulation
23: start the simulation (using play())
```

2.3 Chemistry

In this incarnation, the concentration is an Integer number, representing the number of molecules. Though a specific language targeting morphogenesis is in development, no language is available yet.

2.4 ListDouble

This incarnation is pretty much an experiment and its usage is deprecated. However, it can handle pretty complex scenarios, though not allowing for the same expressiveness of the SAPERE incarnation. On the other hand, it's way faster, since no template matching is involved. No specific language has been implemented yet, and no one will likely be ever implemented.

2.5 SAPERE

2.5.1 Specific language

A Domain Specific Language for SAPERE have been developed. It was realised using the Xtext framework, and as a consequence an Eclipse product with syntax highlighting, code suggestion and automatic code generation is available.

SAPERE DSL installation

In order to be able to write new simulations through the high-level language, the user should download and install the Alchemist-SAPERE Eclipse plug-in. The following instructions do no apply for those who have already downloaded and installed the whole Alchemist package for developers. In fact, they can just run SAPERE DSL environment from within Eclipse (as shown in the video).

For the users that do not need to install the whole development infrastructure, the best way to get the advanced editor and compiler is by installing it as a plug-in in Eclipse. To do so, the user can rely on the Eclipse feature installer, following the instructions available in this video: `https://www.youtube.com/watch?v=0AvzC9rGfpc`. Once installed, the user should apply the Xtext nature to her project in order to enable code highlighting, auto completion and XML generation. Creating a file with extension "alsap" will do the job.

The produced XML file can be opened in Alchemist standalone directly. Who is driving the simulator via Java can interpret the file using the EnvironmentBuilder in order to produce a new IEnvironment.

Encoding and other platform-specific issues

When using the Xtext-generated Eclipse environment in order to produce Alchemist's XML files from the domain specific languages, ensure that the

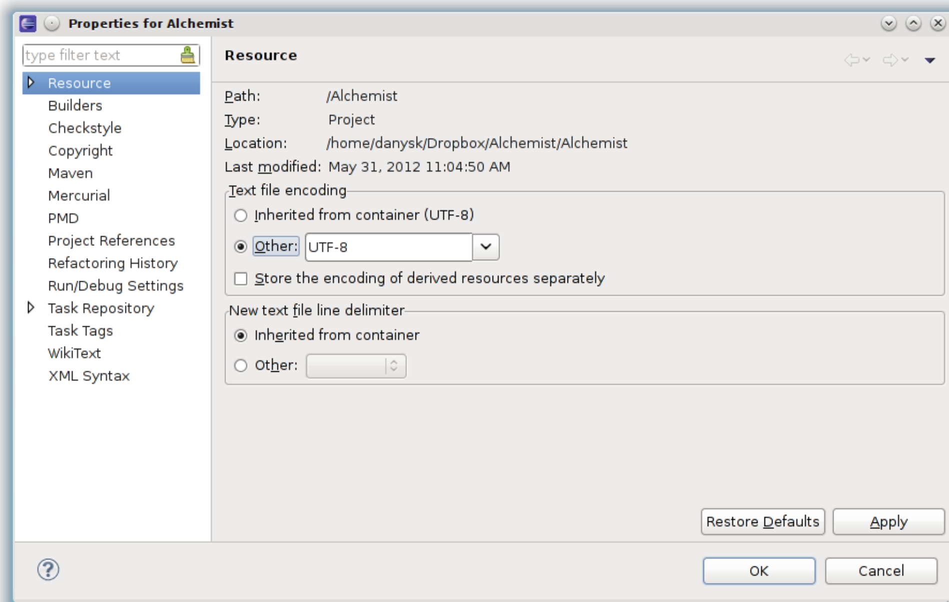


Figure 2.1: The correct configuration for the encoding. The smartest way to set it is to force UTF8 for the main project, and force al the subprojects to inherit it.

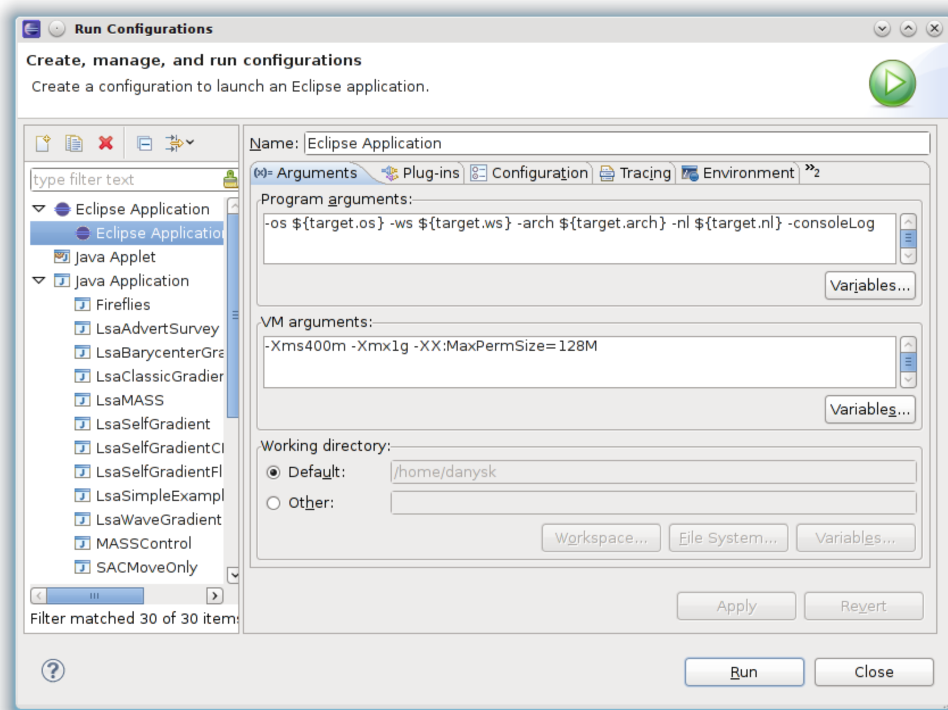


Figure 2.2: The correct configuration for the Eclipse product configuration

option `-XX:MaxPermSize=128M` is passed to the Eclipse product launcher parameters as in Figure 2.2. Also, please check that you're using UTF8 (and not the evil MacRoman) as encoding for the whole project, or the code generation will fail. The configuration should be the same of Figure 2.1.

What to define

For those who are used to the Xtext syntax, the language is defined in the file available at the alchemist source code repository.

Due to Xtext constrains, the simulation files must have extension *alsap*. Each file defines one and one only environment, namely a scenario to simulate. The elements which can be defined are:

- environment (mandatory)
- linking rule (mandatory)
- lsas
- nodes
- reactions
- actions
- conditions

Environment

Let's take a look to a well-formed environment:

Listing 2.1: Use of environment

```
1 environment env type AnEnvironment params "some,parameters,2"
2 with linking rule ALinkingRule params "rule,paramenters"
3 with concentration type ConcentrationType
4 with position type PositionType
5 with random seed 0
6 with time type TimeType
```

Let's analyse it token by token:

1. `environment`: is the keyword signalling the beginning of our specification
2. `env`: is the name of the environment. It's optional.

3. `of type AnEnvironment params "some,parameters,2"`: we want to istantiate an IEnvironment whose real Java class is `AnEnvironment`, using a constructor accepting three paramenters "some,parameters,2". `type` must be followed by a class name which must be implemented in the package
it.unibo.alchemist.model.implementations.environments. An alternative to this line is to specify `default environment`, which tells the system to fall back to Continuous2DEnvironment. This default environment is a continuous space where both the coordinates and the distances are defined by double numbers.

4. `with linking rule ALinkingRule params "rule,paramenters"`: we want the nodes to be connected with the policy specified in a class named `ALinkingRule` belonging to the package it.unibo.alchemist.model.implementations.lin whose constructor expect two parameters. It is possible to use a default rule by specifying `linking nodes in range N` (where `N` is a number). If the default is used, the environment will automatically link together nodes within a certain range.

5. `with concentration type ConcentrationType`: is the Java class to use as concentration. Not used normally, it's optional, and if not specified the system falls back to LsaConcentration.

6. `with position type PositionType`: is the Java class to use as position definition. It's optional and defaults to Continuous2DEuclidean. The user may want to change it only if she switches to a discrete environment.

7. `with random seed 0`: it's optional and defaults to a random number. It can be useful if the user wants to enforce reproducibility on the experiment: if unspecified, every new interpretation of the generated XML file will produce a simulation whose result will differ from the others. In future, the randomness management will be moved inside the engine itself, and thus this part of the language will probably be removed.

8. `with time type TimeType`: is the Java class to use as Time. It's optional and defaults to DoubleTime. The user may want to use something else just if the double 64bit precision is not fine grained enough for the scenario: there is a loss of precision when you try to sum very high numbers and very low numbers, due to the internal double representation.

Examples:

```
default environment
linking nodes in range N
```

It's the minimum allowed definition for an environment. This creates an environment of type Continuous2DEnvironment, which links nodes which are at most N distance units far.

```
environment of type InfiniteHalls params "10"
linking nodes in range N
```

Builds an environment loading the class InfiniteHalls and passing as the only parameter 10 (it's the hall size), which links nodes which are at most N distance units far.

Lsa patterns

Let's take a look to some well-formed lsa patterns:

```
lsa source <source, gradient0, 0>
lsa allSources <source, T, 0>
lsa withDescription <Var, 0, atom> description "Some text"
```

Those declarations (`lsa` can be used multiple times) assign a name to the LSA which follows. From now on, it's allowed to build reactions and define nodes content using the names instead of writing a zillion times the same LSA.

The LSAs in ALCHEMIST are just plain tuples, à la Linda. Though not as expressive as the D1.1 language, the SAPERE language currently in ALCHEMIST allowes to:

- Use atoms, typing lower case starting strings e.g. `<some, atoms>`
- Use numbers e.g. `<5, 7.254>`
- Use variables, typing upper case starting strings e.g. `<Two, Variables>`
- Use operators on numbers, both binary (sum, subtraction, multiplication, division, minimum and maximum) and unary (module), e.g. `<sum, X+Y>`. In order to use multiple operators in the same element of the LSA, it's required to enclose in round brackets the two variables which belong to the operator, e.g. `<complexOp, (X*5)+(Z-A)>`. Brackets are **required**: things such as `<complexOp, X+Y+Z>` will generate errors. When writing reactions, the usage of operations is sound only on the right side.

- Use comparators on variables. Those only make sense on the left side of the reactions. It's possible to express majority, minority and equality: `>, <, =, !=, <=, >=`. The syntax is `def: Variable OPERATOR Variable`, e.g. `<tuple, A, def: B<A>`.

- Use sets. It's possible to define a set of elements by enclosing them into square brackets, e.g. `<set,[a;b;C;D;1;2;]>`.

- Use operators on sets. As the operators on variables, they only make sense on the right side of a reaction. It's possible to: search the maximum and minimum element in the set (using `min(Set)` and `max(Set)`), add or delete an element to the set (using `Element add Set` and `Element del Set`).

- Use comparators on sets. They can be used only on the left side of a reaction. It's possible to check if some set is empty or not, or if some set is contained in another set or not. The syntax is, respectively: `def: Set isempty`, `def: Set notempty`, `def: Set has [some;elements;]`, `def: Set hasnot [some;elements;]`. In order to compare with a single variable, the user must enclose it in a fake set, e.g. `def: Set has [Variable;]`.

- Attach a description. A description can be whatever text string, including every UTF-8 character. It will be always considered as an atom, even if it contains numbers, operators, comparators or starts with an uppercase. It will be inserted as last argument of the tuple: this must be considered when manipulating it through the normal eco-laws: the example LSA provided, in fact, will have four arguments, not three. This hook is useful, in particular, if you need richer items than the provided flat tuples. For instance, if you need to attach semantic information, you can encode entire RDF specifications, and manipulate them in the simulator by writing a specific Agent.

Nodes

The language offers facilities to create a single node in a specific point of the space or multiple similar nodes at once in a rectangular or circular area. Let's analyse an example:

```
place node
named nodename
of type NodeType
at point (0,0)
```

```
containing anLSA <another,one>
with reactions
[] --> []
[] --> []
```

- `place node`: Creates a single node.
- `named nodename`: Assigns a name to the node. It's optional.
- `of type NodeType`: Uses a specific node type, loading the Java class which is passed. The class must be part of the package `it/unibo.alchemist.implementations.nodes`. This is optional, and the system defaults to LsaNode.
- `at point (0,0)`: it's the point in a bidimensional space where the node will be placed. Mandatory.
- `containing anLSA <"another,lsa">`: it's possible to specify which LSAs should be within the node. It's possible to both refer an LSA pattern previously defined or to write new LSAs.
- `with reaction`: after this token, the definition of the reactions begins.

In this second example, we place multiple nodes inside a circle:

```
place 10 nodes in circle (0,0,100)
containing
in all anLSA
```

This specification places 10 nodes randomly in a circle whose centre is in the point (0,0) and whose radius is 100. As for the previous example, it's possible insert a default content in the nodes. The `in all` spatial constrain inserts the LSAs in all the nodes of the group. It's possible to use also the `in rect (X,Y,Width,Height)`, which will inject the LSA only in those node whose position is inside the given rectangle. The last available constrain is `in point (X,Y)`, which only inserts the LSA in the nodes of the group which are exactly in the `X,Y` position

In the third and last example, the nodes are placed inside a rectangular area:

```
place 10 nodes in rect (0,0,2,3) interval 1 tolerance 0.2
containing
in all <all>
in rect (1,1,1,1) <smallRect>
in point (0,0) <origin>
in rect (0,1,0.5,10) <tallRect>
```

The most interesting part is `interval 1`: If specified, the system tries (best effort approach) to separate nodes by a fixed distance. Otherwise, nodes are spread randomly. `tolerance 0.2` forces the system to use some randomness when placing nodes: in particular, it will accept coordinate values which are within ±0.2 distance units from the point they would have been placed if no tolerance would have been specified. Specifying a tolerance turn very useful when an almost-uniform coverage of an area is required, but a total regularity is not desired. `rect (0,0,2,3)` specifies a rectangular region whose lower left point is (0,0), width is 2 and height is 3. Multiple spatial constrains are defined: they allow the user to shape the initial node content in a flexible way.

Eco laws

Let's continue in our learn by examples:

```
eco-law lawName
reaction ReactionType params "some,parameters"
[] --> []
```

- `eco-law lawName`: optional keyword which allows to specify a name for the eco-law. Might be useful in order to render the code clearer.
- `reaction ReactionType params "some,parameters"`: uses the class `ReactionType` from package. `it/unibo.alchemist.implementations.reactions`, calling a constructor with the parameters wrote after `parameters`. It is optional, and the system falls back to LsaExpTimeReaction. Overriding the default class is often useful, in order to create non-exponential timed reactions such as triggers and reactions which happen at specific time ticks. For an overview of the available classes, see the package it.unibo.alchemist.model.implementations.reactions.
- `[] --> []`: this is the core of the reaction. Inside the left brackets, the conditions are specified. Inside the right brackets, the actions are specified. The arrow is just a separator.

```
at time 2 [] -1-> []
```

If the default class is used, the user can specify at which time it should be enabled. Moreover, the rate can be specified directly inside the arrow (similarly to chemical reactions). No rate specified (`-->`) means ASAP reaction. Be careful, since ASAP reactions keep being triggered until their conditions are valid.

```
[<number,A>,<number,B>] -A*B-> []
```

It is allowed to write rate functions. They can turn useful in order to bind the speed of a reaction to some dynamically changing parameter. In the example above, the higher A and B, the faster the reaction. It's possible to write arbitrary complex expressions, using the same language already explained in Section 2.5.1.

Conditions and Actions

Conditions and actions are, respectively, what enables some reaction and what the reaction, when executed, does.

```
[definedLSA] --> [<L,5,a>]
```

Just writing some LSAs inside the brackets (either previously defined or brand new) instantiates a default condition or action. These work in a very similar way to chemical reactions, removing from the LSA space what's on the left side and adding what's on the right side. In this case, `definedLSA` will be removed and `<"L,5,a">` will be added. Note that if `definedLSA` does not contain a variable `L` which the system can match with the `L` on the right side, the execution of this reaction will generate errors.

```
[definedLSA, +<L,S,a>] --> [<L,5,a>]
```

Of course, multiple LSAs can be listed, both on the left and right side. This reaction is similar to the previous, but it only triggers if a tuple `<"L,S,a">` is present in the neighbourhood. The heading `+` means that the condition must be true for at least one node in the neighbourhood. The LSA `<"L,S,a">` will be removed from such neighbour. If we wanted not to remove it, we should have added `+<"L,S,a">` on the right side.

```
[<time,T>] -1-> [<time,#T>]
```

In this example, the tuple time is kept up to date. The trick is the special value `#T`, which is bound to the current simulation time. This one, as every other special value (all those starting with `#`), can be used only on the right side.

```
[<field,Val,Orientation>] --> [*<field,Val+#D,#O>]
```

The `*` character heading an action means that it's executed on all the neighbours. It's the standard way to implement diffusions. This example shows also the other special values: `#D`, `#O`. They all are valid only for reactions which involve the neighbourhood. The first measures the distance between the node and the current neighbour, the second is a vector (in the form `"[X,Y]"`) which points to the current node from the neighbour, namely if the neighbour moves its position of (`X`,`Y`), it will reach the node. The third is a list of the neighbours of the node. In case of actions on multiple neighbours, those values are re-instanced for every neighbour, ensuring a consistent execution.

```
[<"hotBall">] --> [+<"hotBall">]
```

The `+` header can be used for the actions too. It means that the action involves only a (random) neighbour, and not them all. It resembles the point-to-point diffusion system in SAPERE D1.1.

```
[condition ConditionClass params "some,params"] -1->
[agent AgentClass params "ENV,NODE,REACTION,RANDOM"]
```

Both conditions and actions can be told to load specific Java classes, from the packages it.unibo.alchemist.model.implementations.conditions and it.unibo.alchemist.model.imple respectively. It is particularly useful to define mobile agents, whose behaviour might be application (and, thus, simulation) specific. Implementing an agent is reduced to the implementation of a Java class extending LsaAbstractAction, which can then be used inside the language. When the explicit class declaration is used, the special values `ENV`, `NODE`, `REACTION` and `RANDOM` can be used in order to refer to, respectively: the current environment, the current node, the current reaction, the current RandomEngine. That way, it's allowed to build agents which need references to the environment (e.g. for moving or knowing the neighbourhood), to themselves (node), to their behaviour (reaction) and that can take non-deterministic actions (relying on the passed RandomEngine).

Complete example

In Listing 2.2, a grid is built and a gradient is spread. In Listing 2.3, a similar grid is built, but an information about crowd is placed inside an area, creating a sort of "hole" in the gradient. An agent which raises the gradient is created, and it's emergent behaviour is to deflect to avoid the crowded area.

Listing 2.2: SAPERE-DSL example 1

```
default environment
linking nodes in range 1.2
with random seed 1

/*
* defining LSAs for gradient
*/
lsa source
 <source, Type, Time>
lsa target
 <source, target, 0>
lsa gradient <grad, Type, Distance, Time>
lsa gradtemp <grad, Type, Da, T>
/*
* creating a grid of nodes with reactions for gradient
*/
place 42 nodes in rect (0, 0, 6, 7) interval 1
with reactions
eco-law diff [gradtemp] -10-> [gradtemp,*<grad,Type,Da+#D,T>]
eco-law youngest [gradtemp,<grad, Type, Db, def: Tb<T>] --> [gradtemp]
eco-law shortest [gradtemp,<grad, Type, def: Db>=Da, T>] --> [gradtemp]
/*
* creating the node source with the pump reaction
*/
place node at point (6,2.5)
containing target
with reactions
eco-law pump [source] -1-> [<source,Type,Time+1>,<grad,Type,0,Time>]
[gradtemp] -10-> [gradtemp,*<grad,Type,Da+#D,T>]
[gradtemp,<grad,Type,Db, def:Tb<T>] --> [gradtemp]
[gradtemp,<grad,Type,def:Db>=Da,T>] --> [gradtemp]
```

Listing 2.3: SAPERE-DSL example 2

```
default environment
linking nodes in range 1.2
with random seed 1

/*
 * defining LSAs for gradient
 */
lsa source <source, Type, Time>
lsa target <source, target, 0>
lsa crowd <crowd, 10, 0.5>
lsa ctemp <crowd, Level, K>
lsa graddiff <grad, diff, Type, D, T>
lsa gradctx <grad, ctx, Type, D, T>

/*
 * creating a grid of nodes with eco-laws for gradient
 */
place 63 nodes in rect (0, 0, 8, 9) interval 1
containing
in rect (3,3,2,2) crowd
with reactions
[graddiff] -10-> [graddiff,*<grad, ctx, Type, D+#D, T>]
[graddiff,<grad, diff, Type, D2, def: T2<T>] --> [graddiff]
[graddiff, <grad, diff, Type, def: D2>=D, T>] --> [graddiff]
[gradctx,ctemp] --> [<grad, diff, Type, (D)+(K*Level), T>,ctemp]
[gradctx] -100-> [graddiff]
```

```
27
28 /*
29  * creating the node source with the pump eco-law
30  */
31 place node at point (8,3.5)
32 containing target
33 with reactions
34 [source] -0.01-> [<source,Type,Time+1>,<grad, diff, Type, 0, Time>]
35 [graddiff] -10-> [graddiff,*<grad, ctx, Type, D+#D, T>]
36 [graddiff,<grad, diff, Type, D2, def: T2<T>] --> [graddiff]
37 [graddiff, <grad, diff, Type, def: D2>=D, T>] --> [graddiff]
38 [gradctx,ctemp] --> [<grad, diff, Type, (D)+(K*Level), T>,ctemp]
39 [gradctx] -100-> [graddiff]
40
41 /*
42  * creating a person looking for the source
43  */
44 place node at point (0,3.5)
45 with reactions
46 []-10->[agent LsaAscendingAgent params "REACTION,ENV,NODE,graddiff,3"]
47 eco-law decay [<grad, State, Type, D, T>] --> []
```

Notes

This section contains informations that have no space in the previous sections. I expect most of them to be just a "beware of common errors".

1. **Be careful with names** – the Alchemist XML uses names internally to allow the EnvironmentBuilder to build Java references. When names are manually written, those will be used internally. If the same name is used twice, expect something to go wrong, possibly without any error, notice or exception thrown.

2.5.2 Writing agents

Since the agents are normally very simulation dependent, no facility is included in the language to define them, so their implementation must be done in Java. The agent code should implement the interface IAction, and the behaviour should be encoded inside the `execute()` method.

The recommended way to create new agents is by means of extending the existing support classes, namely SAPERELocalAgent, SAPERENeighborAgent and SAPEREMoveNodeAgent. All these classes provide both an implementation of the methods which are required by the simulator internals and some useful methods meant to ease the development of the `execute()` method. SAPERELocalAgent should be used in order to create an agent that operates locally. It provides four constructors, requiring a different number of molecules to be passed. When extending this class, the user must use the proper constructor, passing the templates of the molecules which will

be modified in order for the agent to be correctly interpreted by the engine. SAPEREMoveNodeAgent is specifically targeted to those who need to move the local node, since it provides the `move()` method along with utilities to get the current position (`getCurrentPosition()`) and access the neighborhood. Finally, the SAPERENeighborAgent class provides support for defining agents which write informations onto neighbours.

A nice way to implement more complex agents is to extend AbstractAction, which provides some generic mechanisms already implemented internally. It's recommended to have a look to the existing subclasses of AbstractAction to get familiarity with the usage of the object composing the ALCHEMIST computational model. A particularly important notice is to care of the getContext() method. It is important to return the correct Context: a Context too wide will make the simulation slower (even of magnitude orders), a Context too strict will make the dependency graph to be wrong, and consequently the simulation to have errors. If the agent just wanders around (using the `moveNode` method of IEnvironment and grabs informations from itself and the environment, the Context.LOCAL context is correct (and fast). If the agent is rather complicated and modifies the LSA space content, then the user must remember to add the LSA template it is modifying to the list of the outgoing dependency. If the agent extends from AbstractAction, it should be done simply calling `addModifiedMolecule(yourLSATemplate)` inside the constructor. The superclass will take care of the dependency stuff. If the agent injects or modifies LSAs in the neighbourhood, then the user must also remember to return Context.NEIGHBORHOOD in the `getContext()` method. Finally, if the agent messes up with the LSAs in nodes which are not neighbours (in a SAPERE system, however, this should never happen), then the Context.GLOBAL is required.

2.5.3 Built-in fast gradient for SAPERE

After we realised some case studies with ALCHEMIST, we soon realised that the gradient is a fundamental pattern, sometimes used in order to build more complex spatial structures. In order to maximize the performance, ALCHEMIST provides an heavy optimized and reliable gradient implementation within SAPEREGradient. This class can be instanced from the DSL: an example is given in Listing 2.4.

Listing 2.4: SAPEREGradient usage example

```
1 [...environment definition...]
2
3 lsa s <source, Type, Dist>
4 lsa g <grad, Type, Dist>
5 lsa c <crowd, L>
```

```
6
7 [...node group definition...]
8
9 reaction SAPEREGradient params "ENV,NODE,RANDOM,s,g,2,((Dist+#D)+(0.5*L)),c
      ,200,1" [] --> []
10
11 [...other reactions, nodes, and so on...]
```

This specification defines a new gradient reaction whose source is defined in the fourth parameter. In this case, it references `s`, so LSA template for the source is associated to `<source, Type, Distance>`.
The fifth parameter is the LSA template representing the form the gradient will have. All the variables must be also present in the source template, in order for the gradient LSA to be correctly instanced. In this case, we reference to `g`, which has the form `<grad, Type, Distance>`.
The sixth parameter tells the reaction in which position put the gradient computation. In this case we choose the third element (the numbering goes the same way of the arrays, so we must specify `2`). This means that, when computing the gradient, the `Dist` variable will be substituted with a different value.
The seventh parameter explains which is this value. This must be a correct expression, arbitrarily complex, which can only include variables present in the gradient template and in the context template, plus the synthetic variables (`#D` and `#T`).
The eight parameter defines a context LSA template. The variables included there can be used to influence the computation of the new value. In this case, it refers to `c`, which is defined as `<crowd, L>`. This means that `L` can be used in the formula defining how to compute the new values. It is obviously possible to make a gradient which does not need to be contextualised. In this case, it is safe to write `null` in place of the LSA template name.
The ninth parameter defines a threshold for the gradient: if the computed value goes above this threshold, the gradient is not spread. It can be used to avoid flooding.
The tenth and last parameter defines the speed (Markovian rate) of the reaction. It should be tuned depending on the performance you want to achieve. The higher this value, the more reactive will be the gradient to changes (and the harder for ALCHEMIST to simulate, clearly).

2.5.4 Gradient and Force-based model for Pedestrian Dynamics

Nowadays it is very important to have models to predict the behavior of people in certain circumstances; therefore, it is essential to have a model for

the simulation of realistic pedestrians. A realistic model is the force-based model, in which the choice of the direction to follow and the interactions are modeled by means of two forces [11]:

- Attractive force represented by the target.
- Repulsive forces represented by other pedestrians and obstacles that the agent meets in own way.

Speed Computation

To calculate the pedestrian speed we consider four forces: desired force, social force, dodge force and obstacle force. At each execution the acceleration is calculated as weighted sum of these forces and, subsequently, the new speed is obtained from the previous step speed and the current acceleration. This value must be obviously lower than the maximum allowed speed of a pedestrian in order to simulate the different speeds that a pedestrian can support.

Desired Force component The desired force represents the attractive force to the target; the target is the node in the pedestrian's neighborhood which has the lowest gradient value. The components of this force are proportional to the distance from the target node, according to the following formulas:

$$desiredForce_x = \frac{distance_x}{distance}$$
$$desiredForce_y = \frac{distance_y}{distance}$$

Where $distance_x$ and $distance_y$ are respectively the distance along the x-axis and the y-axis between the target node and the pedestrian; while *distance* is the Euclidean distance between these nodes.

Social Force component The social force is the repulsive force among pedestrians which ensures that they will avoid each other without colliding.

According to [9] and [14], every human being has the need for a living space that, under normal conditions (i.e. not panic), is required to be not violated. If one pedestrian j invades the vital space of another pedestrian i, namely he is at a distance lower than a certain interaction range (rappresented by the variable *INTERACTION_RANGE* in class *SocialForceAgent*), the repulsive force is calculated according to the following formula [11]:

$$F_{ij}^{\vec{rep}} = -m_i k_{ij} \frac{(\eta v_i^0 + v_{ij})^2}{d_{ij}} \vec{e_{ij}}$$

Where:

- m_i is the mass of $i - th$ pedestrian;
- k_{ij} reduces the effective range of the repulsive force to the angle of vision and is calculated with the following formula:

$$k_{ij} = \begin{cases} \frac{(\vec{v_i} \cdot \vec{e_{ij}})}{\|\vec{v_i}\|}, & \text{if } \vec{v_i} \cdot \vec{e_{ij}} > 0 \text{ and } \|\vec{v_i}\| \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

- η is the social force strength; we have chosen a value equal to 0.2 according to [11].
- d_{ij} is the effective distance between pedestrian i and j.
- v_{ij} is the relative speed and it is defined in such a way that slower pedestrians are less affected by the presence of faster pedestrians in front of them:

$$v_{ij} = \begin{cases} (\vec{v_i} - \vec{v_j}) \cdot \vec{e_{ij}}, & \text{if } (\vec{v_i} - \vec{v_j}) \cdot \vec{e_{ij}} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- e_{ij} is the force direction between the two pedestrian.

Dodge Force Component The dodge force represents the force which enables a pedestrian to turn to the right or to the left if she is at a distance shorter than a certain range (rappresented by the variable *PROXIMITY_TURN_RANGE* in class *SocialForceAgent*) from another pedestrian. This distance was set to 0.5 according to [25]. According to [15], depending on the country in which they are grown, people prefer to get around obstacles from the right or from the left: for example, the British (according to their driving rules) prefer to get around obstacles from the left, while in the rest of Europe from the right. In particular, our solution is to make a change of direction of ±45 degrees according to the following mathematical formulas:

- Turning to the left:

$$dodgeForce_x = -dodgeForceStrength * desiredForce_y$$
$$dodgeForce_y = dodgeForceStrength * desiredForce_x$$

- Turning to the right:

$$dodgeForce_x = dodgeForceStrength * desiredForce_y$$
$$dodgeForce_y = -dodgeForceStrength * desiredForce_x$$

Where $dodgeForceStrength$ is a factor that represents the force intensity. A factor (represented by the variable *TURN_RIGHT_PROBABILITY* in class *SocialForceAgent*) represents the percentage by which a pedestrian choose to turn to the right. This factor is, obviously, higher for Europeans and lower for Asians and British.

Obstacle Force component Finally, the obstacle force represents the repulsive force generated by the obstacles encountered along the way. Given all the obstacles that the pedestrian can see, which are all those within a certain range (represented by the variable *OBSTACLE_INTERACTION_RANGE* in class *SocialForceAgent*), the nearest is taken and the repulsive force is calculated in the center of this obstacle. Due to the fact that we have choose to calculate the force in the center of the obstacle, the pedestrian is subject to a force that is bigger as the pedestrian is far from the center; in this way we have a correct contribution even if the obstacle has not regular shape (square) but irregular like rectangle. Moreover, only the nodes which are in sight - namely, those which are not behind an obstacle - are considered in order to compute the available directions.

$$obstacleForce_x = -obstacleForceStrength * \frac{distance_x}{distance}$$
$$obstacleForce_y = -obstacleForceStrength * \frac{distance_y}{distance}$$

Where $obstacleForceStrength$ is a factor that represents the force intensity; $distance_x$ and $distance_y$ are the distance along the x-axis and the $distance$ along the y-axis respectively between the center of the obstacle and the pedestrian and distance is the Euclidean distance between these nodes.

Displacement computation

Once the speed is calculated as the weighted sum of the four forces described above, the displacement is obtained from the variation of the pedestrian position in the unit of time Δt, using the following formulas:

$$dx = \Delta t * v_x$$
$$dy = \Delta t * v_y$$

Where v_x and v_y are the speed vector components.
To avoid the occurrence of sudden changes in direction, the final displacement is calculated as a weighted average between the displacement calculated

at the previous cycle and the displacement computed in the current cycle. In particular, the displacement of the previous cycle will have a greater weight (80%) in order to not modify too much the desired direction of the pedestrian, namely to not have sudden direction changes. If the pedestrian is in front of another pedestrian and the latter is sufficiently close to the first, a displacement adjustement is performed that is equivalent to a slowdown.

Reach the target

Because not all pedestrians can exactly reach the target, we chose to make them stop only if, for a certain number of calculation cycles (represented by the variable *MIN_DISP_CYC_TH* in class *SocialForceAgent*), their displacement is lower than a certain value (represented by the variable *MIN_DISPLAC EMENT* in class *SocialForceAgent*). This means that the pedestrian can not proceed further (for example due to the presence of other pedestrians) and so he has to stop in the reached position because this is surely the closest position to the target node; if there is another target to reach, it is possible to reset the flag (represented by the variable *targetInLineOfSight* in the class *SocialForceAgent*) that represents the reaching of the target in order to make pedestrians move again. The minimum displacement (*MIN_DISPLACEMENT*) is 0.01; this value has been chosen after verifying that when pedestrians are around the target, they aren't able to perform displacement with a value greater than 0.01. Greater values may cause pedestrians stop before they've reached the target. Moreover, the value of the threshold (*MIN_DISP_CYC_TH*) can be choosen in the range [150;250], these values derive from several tests in which the time necessary to stop hs been measured (see the table Table 2.1).

Table 2.1: Minimum displacement cycles threshold

Value	*Runs*	*Average time*
150	5	1.38s
200	6	1.75s
250	4	2.40s

Smallest values make the convergence time decrease but in many cases make pedestrians to stop too far away from the target node; the same thing appens, more rarely, when 150 is chosen as value for the parameter. This

make us choose 200 as best trade-off between the convergence time and distance from the target.

Finally, the user can decide, by setting a boolean flag (represented by the variable *stopAtTarget* in class *SocialForceAgent*) at pedestrian creation, if pedestrians have to stop the movement once the target is reached. If this flag is true pedestrians will stop as discussed above, otherwise they will continue to move: this could be useful in the case that the user want to add a new target node as the previuous is been reached.

Group movement

Each pedestrian has a groupId that is equal to 0 if the pedestrian does not belong to any group and is different from 0 if the pedestrian belongs to some group. The realization of the group behavior was obtained by using a kind of positive reinforcement through the release of *pheromone* modelled by a tuple like *pheromone(N, GroupId)* where *N* is the amount of pheromone in that node and *GroupId* is the identifier of the group that must follow that pheromone. Initially pedestrians search the node with minimum gradient and release pheromone in it. In subsequent cycles, each pedestrian search the node that has the highest pheromone value and moves towards it. Once the pedestrian reaches that node, each pedestrian searches again the node with the lower gradient value, releasing again pheromone; this described behavior is repeated cyclically. To avoid that pedestrians do not continue, seeking the node with higher pheromone value, to choose the same node, it is denied to return to nodes that are already visited. You can see that in the group behavior there is a modest presence of randomness, in fact, if there are two nodes with high levels of pheromone, it may happen that two different groups emerge. This is a desired behaviour to recreate a random behavior for which a pedestrian can choose not to follow the rest of the group.

Qualitative evaluation

The realistic pedestrians have been qualitatively evaluated during the AlmaOrienta 2013 event in Bologna. We took a few hours of videos of people walking the exhibition, then we rebuilt the environment within the simulator, placing the realistic pedestrians in positions resembling those filmed. We tried to let the pedestrian move and see if their movement was qualitatively comparable to what was on the videos. We obtained good results, in particular we have seen a similar behaviour in avoiding other people (the videos confirmed the preferentially avoidance towards the right hand side), in group behaviour (pedestrian belonging to big groups tend to split in sub-groups of

few units each) and in dealing with physical obstacles, in particular doors.

How to use

To use this Alchemist extension is necessary to define an environment populated with nodes that represent pedestrians. In this section will be illustrated an example of usage.

A simple example In Listing 2.5, a simulation file is provided featuring:

- One target
- A set of European pedestrians who want to reach a target
- A set of pedestrians who stay in the center of the environment

Listing 2.5: Usage example for the realistic indoor pedestrian model

```
1  default environment
2  linking nodes in range 1.5
3
4  lsa source <source, Type, Distance>
5  lsa target <source, target, 0>
6  lsa gradient <grad, Type, Distance>
7  lsa crowd <crowd, L>
8
9  place 200 nodes in rect (0,0,19,9) interval 1
10 containing in point (16, 4) target
11 with reactions
12 reaction SAPEREGradient
13 params "ENV,NODE,RANDOM,source,gradient,2,((Distance+#D)+(0.5*L)),crowd
       ,2000000,10" []-->[]
14 eco-law compute_crowd []-1-> [agent CrowdSensor params "ENV,NODE"]
15
16 place 15 nodes in circle (3, 4, 3)
17 containing in all <person>
18 with reactions
19 []-100->[agent SocialForceEuropeanAgent params "ENV,NODE,RANDOM,gradient
       ,2,0,false"]
20
21 place 20 nodes in circle (10, 4, 2)
22 containing in all <person>
```

The execution rate was setted to 100 supposing that at each execution a time interval (represented by the variable *DELTA_T* in class *SocialForceAgent*) of 0.01 is used to compute the new displacement. This means that in one second a time interval of 1 is considered. Changing the execution rate value lead to change the time interval value too.

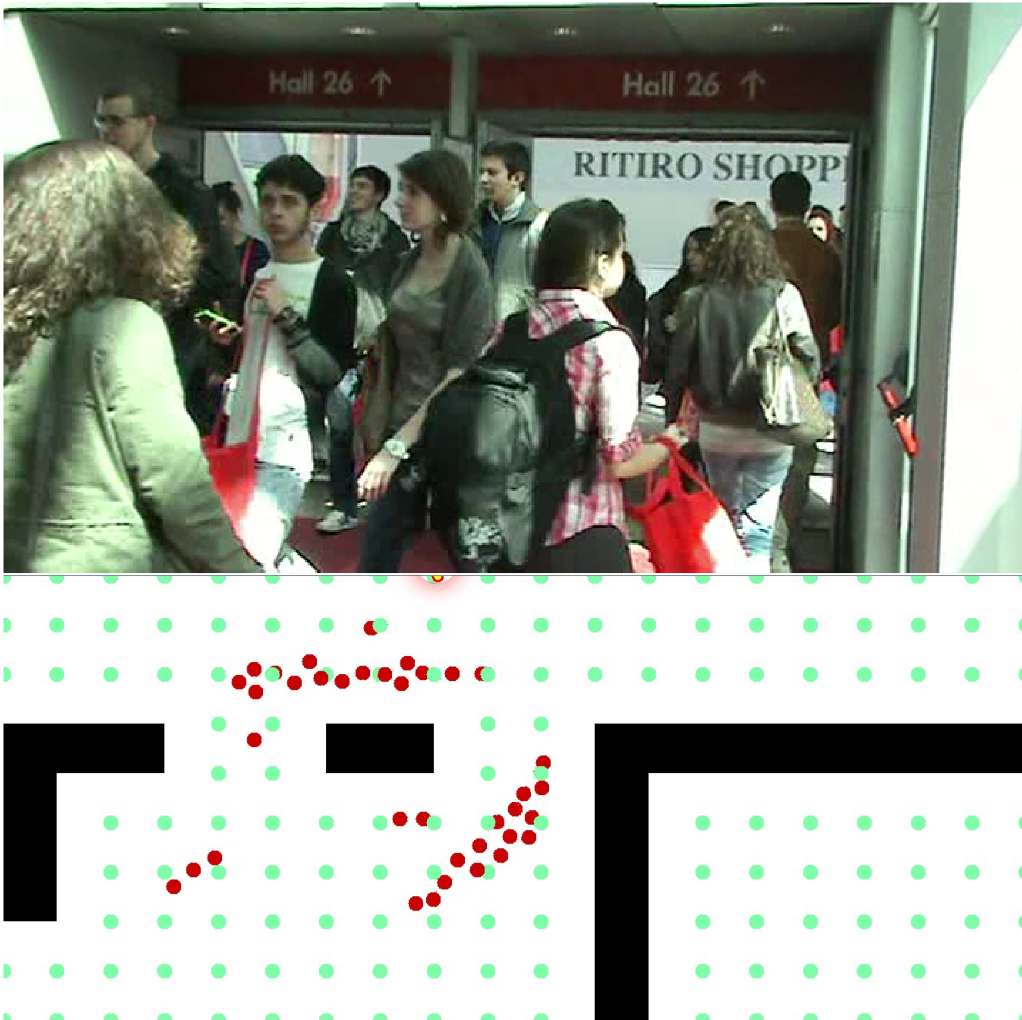


Figure 2.3: A video frame took during the event, and a snapshot of the simulator running the same two-groups passing by the two doors.

Stopping at the target To make pedestrians to stop or continue the movement once the target has been reached is sufficient to change the sixth parameter of the constructor of the pedestrian agent. To make pedestrians to stop set the sixth paramenter to false, otherwise set it to true.

Group behavior To make pedestrians move towards the target forming a group is sufficient to change the fifth parameter of the constructor of the pedestrian agent. In fact, if you specify the value zero as in the previous paragraph the pedestrian is not part of a group. If you specify a number not equal to zero, the pedestrian is part of the group corresponding to that number and it moves towards the target joining the pedestrians of the group. Listing 2.6 presents an example of group specification:

Listing 2.6: Usage example for the realistic pedestrians group dynamics

```
1  [...environment definition...]
2
3  [...lsa definition...]
4
5  place 15 nodes in circle (3, 4, 3)
6  containing in all <person>
7  with reactions
8  []-100->[agent SocialForceEuropeanAgent params "ENV,NODE,RANDOM,gradient
       ,2,1,false"]
9
10 [...gradient definition...]
11
12 [...other reactions, nodes, and so on...]
```

Two pedestrian sets To define two sets of pedestrian having two different targets is necessary to define a second target and a second code block representing another set of pedestrians. See in Listing 2.7 an example of such specification:

Listing 2.7: Example involving realistic pedestrians with two gradients

```
1  default environment
2  linking nodes in range 1.5
3
4  lsa source <source, Type, Distance>
5  lsa source2 <source, Type, Distance>
6  lsa gradient <grad, Type, Distance>
7  lsa gradient1 <grad, target, Distance2>
8  lsa gradient2 <grad, target2, Distance2>
9  lsa crowd <crowd, L>
10
11 place 200 nodes in rect (0,0,19,9) interval 1
12 containing
13 in point (16, 4) <source, target, 0>
14 in point (3, 4) <source, target2, 0>
15 with reactions
16 reaction SAPEREGradient params "ENV,NODE,RANDOM,source,gradient,2,((Distance
       +#D)+(0.5*L)),crowd,2000000,10" []-->[]
```

```
17 eco-law compute_crowd []-1-> [agent CrowdSensor params "ENV,NODE"]
18
19 place 15 nodes in circle (3, 4, 3)
20 containing in all <person>
21 with reactions
22 []-100->[agent SocialForceEuropeanAgent params "ENV,NODE,RANDOM,gradient1
     ,2,0,false"]
23
24 place 15 nodes in circle (16, 4, 3)
25 containing in all <person>
26 with reactions
27 []-100->[agent SocialForceEuropeanAgent params "ENV,NODE,RANDOM,gradient2
     ,2,0,false"]
28
29 place 20 nodes in circle (10, 4, 2)
30 containing in all <person>
```

Obviously, to make the two pedestrians sets move towards their target forming a group, is sufficient to specify a value not equal to zero in the fifth parameter of pedestrian agent constructors.

Defining pedestrians of different cultures In a previous section, we said that European and Asian people has different approach to dodge the obstacle. To implement this and other features (like different speed, different pedestrian size and different vital space size) is necessary to write a SocialForceAgent extension and define there, it the parameters that you want to change. For more detail just take a look at the implementation of the following classes:

- *SocialForceAgent*: representing a generic pedestrian agent
- *SocialForceEuropeanAgent*: extension representing the European behavior
- *SocialForceAsianAgent*: extension representing the Asian behavior

2.6 Additional features

2.6.1 Alchemist2Blender: An Alchemist to Blender interface

Alchemist2Blender is a simple Java class which runs a simulation and outputs data - using the Alchemist2BlenderOutputMonitor - to some files, in a csv-like format. These can be later importer into Blender to have a 3D rendering of the simulation.

The Java OutputMonitor The OutputMonitor is very simple: for each step it writes the simulation data to two different files:

- one containing the moving nodes (.nod file extension);
- one containing the gradients (.gra file extension);

It also writes all the obstacles in a .wal file. The .nod file row structure is the sequent: timestamp;step_number;node;node;...; where 'node' is composed by:

- node_name: the node's ID
- node_x: its 'x' value
- node_y: its 'y' value
- node_options: other parameters contained in the node.

For example:
`0,00898;00020;469,8.52764,4.67685,person;468,7.80877,4.59889,person;...`
The .gra file structure is very similar, but 'node_options' is substituted by 'node_z', as they have three dimensions: timestamp;step_number;node;node;...; where 'node' is composed by:

- node_name: the node's ID
- node_x: its 'x' value
- node_y: its 'y' value
- node_z: its 'z' value

An example can be found in the following line:
`0,21576;00300;344,24.0,8.0,3;319,24.0,7.0,2;318,23.0,7.0,3;`
The .wal file is even simpler, as it contains just couple of points x1,y1,x2,y2 which are the opposite vertices of an obstacle. For example:
`33.75,18.5,34.0,20.0;32.5,19.75,34.0,20.0;...`
Alchemist2Blender may be launched with just one parameter, that is the full path of the input XML file. The other two parameters are optional and are the max number of steps to process - default: 2000 - and the frame/step rate - default: 25. This last parameter may be useful in case of slow simulations, as it outputs only one frame every N ones, making the rendering faster.

The Blender interface The Blender interface consists in some Python files which have to be put in the Blender add-on directory; after this, they may be activated from the 'User preferences' menu. When the add-ons are active, in the 'File -¿ Import' menu you may find some new entries, for example 'Alchemist nodes (.nod)'. The user just needs to choose the correct file, and the script does everything else. All the add-ons have a very similar functioning:

- first of all, the module imports data in a Python dictionary;
- then it creates all the meshes which represent the data, for every step;
- afther this, it creates a frame handler wich, for every frame, hides all the objects which are not related to that step.

This last passage is not needed when importing obstacles, as their position does not change during the simulation. Some of the add-ons may need external libraries, which are specified in the module comments, as well as the location where to download the needed files.

2.6.2 From png image to Environment

Scope

Today on the web it's easy to find many png images that represent maps useful to use in some simulations. It's also easy to create a particular map of interest using any painting tool and saving it in png. Than the aim of this extension is creating Environments, usable in the simulator, from png images.

Used algorithms

In the development of this extension were created several algorithms. First we must read the png image and take all the informations we need in the next steps. The algorithm used to extract informations is very simple:

1. Read all the pixels colour values and save them;
2. Individuate all the possible pixels that can be vertices of a shape.

To do the second step the algorithm used, in details, is the following:

Algorithm 5 Individuate all the possible pixels that can be vertices of a shape

```
 1: create an empty list for the possible vertices
 2: for each pixel in the image do
 3:     read the pixel colour value
 4:     if the pixel is the first or the last of a line then
 5:         add the pixel to the list
 6:         save the its colour value
 7:     else
 8:         if the current colour value is different from the previous then
 9:             add the current pixel to the list
10:             add the previous pixel to the list
11:             save the current colour value
12: return the list that contains all the possible vertices
```

The most important and complex algorithm is the one that does the partitioning of the image in rectangle. This algorithm is based on the article [10]. The main steps are:

1. Finds the vertices of the image and controls if are concave or convex;
2. Sorts them by row;
3. Finds the edges of the figures;
4. Sorts the vertices by column;
5. Finds internal lines joining couple of collinear edges;
6. Finds the maximum subset of these internal lines where every line do not touch each other;
7. Completes the partition;
8. Finds all the rectangles.

To implement the first step for every pixel in the list, returned from the Algorithm 5, we analyse the adjacent pixels. Every vertex of the pixel is a candidate vertex of the image. We consider adjacent pixels as you can see in the Figure 2.4.

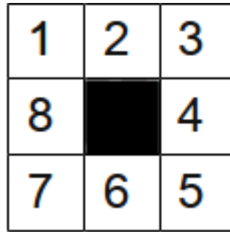


Figure 2.4: Pixels adjacency all the numbered pixels are adjacent to the black one.

When the vertices list is sorted by row the edges of the figure are easy to find.

Algorithm 6 Finds the edges of the image

1: create two empty edge list h-edge and v-edge
2: **for** each vertex of the image **do**
3:     **if** it's not a vertex of a horizontal edge **then**
4:         the next vertex in the list is the other vertex of the horizontal edge
5:         add the founded edge to h-edge
6:     **if** it's not a vertex of a vertical edge **then**
7:         the first vertex in the list with the same y value is the other vertex of the vertical edge
8:         add the founded edge to v-edge

The next step of the algorithm finds the internal line joining couple of collinear edges. An internal line joining two horizontal\vertical collinear lines exists only if:

1. There are four consecutive horizontal\vertical vertices with the same y\x coordinate;

2. The second and the third vertex are concave;

3. No vertical\horizontal edges intersect the segment joining the second and the third vertex.

Now the algorithm proceed finding the subset of internal lines that do not intersect each other. Before applying the proposed algorithm we divide the internal lines with no intersection from the others and then uses the procedure only on the lines that at less intersects another one. Every internal edge that have an intersection can be represented as a vertex of a graph and every edge of the graph exist only if the there is an intersection between the two vertices (the internal edges). Two algorithm have been created to do this step, one finds the maximum number of non intersecting edge that is the optimum solution and the other is a simply heuristic that find a subset. The

implementation of the algorithm that finds the optimum follows the article [20]. A representation of the algorithm can be:

Algorithm 7 Finds the maximum independent set

```
 1: create a empty list of edges called result
 2: create the variable maxindependentsize and set them to zero
 3: while the graph cardinality isn't zero do
 4:     select the edge that has more intersections
 5:     create a list called mis containing all the edges apart the ones that intersects the selected edges
 6:     if the mis size is greater than two and greater than maxindependentsize then
 7:         remove intersecting edges from the list
 8:         if the list size is grater than maxindependentsize then
 9:             maxindependentsize = list size
10:             result = list
11: all the remaining vertices of the graph are independent
12: if the remaining edges number is greater than maxindependentsize then
13:     result = list containing all the remaining edges
```

The heuristic algorithm is faster than the other and easy to explain. We obtain the list of edge that have at least one intersection from the graph, than one edge at time we control if it have intersection with the others edges in the list. When an intersection occurs the edge is removed from the list and the algorithm proceed with another edge. If no intersection are finded we proceed with the following edge.

At the end the lines with no intersection are added to the list created by the algorithm used.

To complete the partition for every remaining concave vertex that isn't a vertex of an internal line we draw an horizontal line until the next vertical edge or internal line.

The last, but not the less important, step of the algorithm is the individuation of all the rectangles thanks to the informations created by the previous steps. The algorithm used here wasn't taken from the literature, it was created expressly for this extension. It's explained by the followings lines of code:

Algorithm 8 Individuate all the rectangles

```
 1: create an empty rectangles list called result
 2: while there are horizontal edges do
 3:     find the top left edge, this is the one or one of the horizontal edges composing the top horizontal
        edge of the rectangle
 4:     try to find the first vertical edge that intersect
 5:     while there aren't vertical edges that intersect do
 6:         find the next horizontal edge
 7:         try to find the first vertical edge that intersect
 8:     the founded edge is the one or one of the vertical edges composing the right vertical edge of the
        rectangle
 9:     try to find the first horizontal edge that intersect
10:     while there aren't horizontal edges that intersect do
11:         find the next vertical edge
12:         try to find the first horizontal edge that intersect
13:     the founded horizontal edge is the one of one of the horizontal edges composing the bottom
        horizontal edge of the rectangle
14:     add the rectangle founded to result
15:     remove from the edges lists all the edges composing the rectangle founded
```

How to use the extension

As seen in Section 2.5.1 you can write new simulations through the high-level language. To use this extension it's sufficient to use as environment in the *alsap* specification file the PngEnvironment. The params that can be given to the environment are the following:

- A double value called range (mandatory);
- A long value called delta;
- The png image path (mandatory);
- A double value called zoom;
- A boolean called findopt;
- A list of colours that represent the obstacles.

The range value is the maximum comunication range between the nodes of the environment;

Delta is the x and y coordinate value offset of the map in the environment. For example if the given delta value is three all the x and y coordinate of the map are augmented of three. If not specified the delta value is by default zero, and consequently no offset is applied.

The image path specified must be absolute.

With the zoom value you can specify the zooming factor of the image in percentage. For example, if you want a fifty percent zooming factor you must pass 50. If not specified by default its value is 100.

The boolean findopt must be true if you want find the optimum result or false otherwise. By default its value is set to false.

Every colour that represents obstacles must be passed as a triple of int values. Any int value is a RGB component of the colour, if no colours are specified, the system defaults to black.

The simplest example you may try is the one in Listing 2.8, while a more complete one can be found in Listing 2.9.

Listing 2.8: Simple example with PngEnvironment

```
1 environment type PngEnvironment params "path/to/your/image.png"
2 linking nodes in range 1
```

Listing 2.9: More complete example with PngEnvironment

```
1 environment type PngEnvironment params "10,/path/to/your/image.png,false"
2 linking nodes in range 75
3
4 lsa s <source, Type, Distance>
5 lsa target <source, target, 0>
6 lsa gradient <grad, Type, Distance>
7 lsa c <crowd, L>
8
9 place 700 nodes in rect (0,0,1500,1200) interval 50
10 containing in point (400, 900) target
11 with reactions
12 reaction SAPEREGradient params "ENV,NODE,RANDOM,s,gradient,2,((Distance+#D)
     +(0.5*L)),c,200000,10" []-->[]
13 eco-law compute_crowd []-1-> [agent CrowdSensor params "ENV,NODE"]
14
15 place 15 nodes in circle (900, 900, 100)
16 containing in all <person>
17 with reactions
18 []-100->[agent SocialForceEuropeanAgent params "ENV,NODE,RANDOM,gradient
     ,2,0,false"]
```

2.6.3 Real world maps

Alchemist is able to load and simulate on real world maps from OpenStreetMap [3]. It is able to navigate in those maps pedestrians, cars and bikes very quickly, thanks to GraphHopper [2].

Preliminary operations

Downloading maps OpenStreetMap allows users to download an arbitrarily big area of the world (whole planet included). Such map is encoded in XML. To obtain a map given some boundaries (the so-called bounding box), an online service is available [7]. When using such service, remember to disable the search by tag (you will need the whole data in the map, then use the "search by area" tool to obtain boundaries. An URL will be displayed, showing the query which should be performed to get the XML file.

Obviously, a browser is not the perfect tool for downloading a hudndred of megabytes big XML file, and consequently I suggest to use `wget` [1] or a similar tool for performing the operation. In case of wget, the command to issue will look like:

```
wget http://open.mapquestapi.com/xapi/api/0.6/*[bbox=MinLong,MinLat,MaxLong,MaxLat]
```

For instance, the following command downloads a map for the whole city of Cesena (Italy):

```
wget http://open.mapquestapi.com/xapi/api/0.6/*[bbox=12.1734,44.11947,12.3107,44.1749]
```

For the most commonly downloaded portions of the globe, specific web services exist that provide updated versions of the map ready for a fast download. A comprehensive list of those services is available at [6].

Refining maps Once downloaded, it is warmly suggested, albeit not mandatory, to refine the map. There are two reasons for executing this operation:

1. Performance: Alchemist can import the OpenStreetMap maps in three forms: XML, compressed XML or Protocolbuffer Binary Format (PBF). The first format is great for ease of parsing and sharing, but the last is far more efficient both in terms of performance and space consumed. To better understand the difference, consider that the same map (we run tests with a map of the whole city of Vienna) was over 200MB in uncompressed XML, 42.6MB in compressed XML and 21MB in PBF. The processing time was over a minute for the first two and below ten seconds for the latter.

2. Exclusion of unnecessary parts: often, downloaded maps include portions of the world we are not interested in, e.g. if a long straight highway or a big part is part of the downloaded file, it may be included entirely, even if the boundaries are outside those manually specified.

To overcome both problems, I suggest to use Osmosis [4]. If you have a map.xml file that you want to convert into a map.pbf file filtering precisely the bounding box with coordinates N, W, S and E (respectively for north, west, south and east) you can issue a command such as:

```
osmosis --rx enableDateParsing=no file=map.xml --bb top=N left=W bottom=S right=E --wb file=map.pbf
```

Details on the usage og Osmosi are available at [5]

IMapEnvironment

The environment designed to support the OpenStreetMap maps is OSMEnvironment which implements IMapEnvironment. The smaller constructor available accepts a single parameter, namely a String with the path of the map. When using this environment, it is strongly recommended not to use the default IPosition type but to explicitly use LatLongPosition. In fact, this class allows for specifying points using latitude and longitude instead of the classic x and y positions, and also automatically provides to translate the distances in meters. When using such positions, also the nodes placement must be specified in latitude/longitude format.

In Listing 2.10 a minimal example is written. Alchemist will load the map in `/home/user/map.pbf`, will use latitude and longitude as positions and will link together nodes which are at most 50 meters away from each other.

Listing 2.10: Loading a OSMEnvironment

```
1 environment type OSMEnvironment params "/home/user/map.pbf"
2 linking nodes in range 50
3 with position type LatLongPosition
```

GPS traces

Available agents

2.6.4 Approximate Probabilistic Model Checker

Approximate Model Checking in a nutshell

This paragraph has the sole purpose to be an hand extended towards the totally APMC unaware user. A more exhaustive introduction can be found in many articles and manuals, like [19], [16], [8].

In the purpose of consistently evaluating the behavior of complex stochastic systems, and considering the typical impossibility to perform proper model checking for such systems, a facility is provided in Alchemist to obtain approximate model checking.

Proper model checking analyzes any possible state of the system, to check properties like liveness or safety. Instead of analyzing any state, which rapidly becomes unfeasible as the number of states grows, an approximate model checker performs many simulations of the observed system and returns not an exact result, as a proper model checker would, but a confidence interval of a certain dimension for the observed property.

An example could be: *With which probability the good thing I expect from my system will happen within 10 seconds?* The answer to this question could be the 99% confidence interval $[0.95, 0.97]$. Not only probabilities, though. Confidence intervals can also be obtained for discrete and real values, answering questions like *how long does it take for...* or *How many items reach the target whitin...*

How to define the property to check

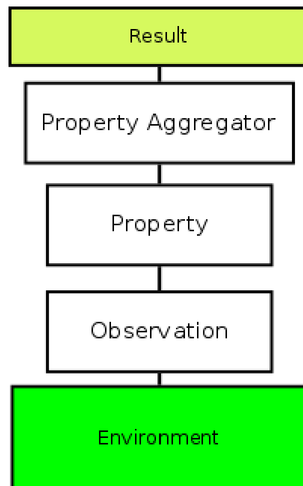
Since a specification language for properties hasn't been built yet, the user must write the needed classes himself, implementing the interfaces specified in package `it.unibo.alchemist.modelchecker.interfaces`.

As shown in figure 2.5, three levels lie between an Environment and the model checking result:

Observation The user must override here the `observe` method, which, given an `IEnvironment` as a parameter, must look into it and verify the property of interest. This can be actually tricky at this time, due to the lack of the possibility of naming things in an environment. Is it possible, however, to exploit the automatic numbering of nodes, and the `getNodeByID` method of IEnvironment. Various aspects can be inspected, like a node's position or the concentration of chemicals. The method `observe` returns the value of interest, that can be of type boolean or any numeric type.

The method `canChange` must be overridden too, and must return `false` when observation can't change, no matter how long the simulation will go on.

Figure 2.5: The three tiers between Environment and model checker



Property This tier allows the user to compose observations in different ways; a property can be the boolean `AND` operation over two (or more) observations, or the sum, or mean, over some non boolean observations. Some classes have been provided to help the programmer: a generic property, the boolean `AND`, the boolean `OR`.

Properties have a `canChange` method too, whit the same meaning of above.

PropertyAggregator This last tier accepts as input the properties of all the pool of simulations and performs the final aggregation; two main cases are the transformation of a number of boolean observations in success probability, or the computation of the mean of many numeric observations. These cases are realized in `EventProbability` and `MeanAggregator`.

In order to exploit some performance boost available in low variance cases it is necessary to be able to compute the variance of aggregated data. `PropertyAggregator` has been extended in `PropertyAggregatorVariance`, which adding the `getS` method provides this functionality.

Performing the model checking

Once one or more `Observations` are built and put into a `Property`, and as a `PropertyAggregator` is ready, a new instance of `AlchemistAPMC` can be created; it's time to specify the desired confidence interval as well.

As the object is constructed, we can get the process started invoking the `execute` method, which requires three parameters: the path to the xml rep-

resentation of the environment, the maximum number of steps and maximum execution time per simulation.

The call to `execute` will immediately return flow control to the invoking thread. To get (and eventually wait for) the result, the `getResult` method has to be called.

On AlchemistAPMC statistics

The reasoning behind this APMC engine is based mainly on [16] and [8]. In these works, different statistical bounds are demonstrated for the number of simulation necessary to obtain the desired confidence interval. These results have been merged obtaining the bound used in AlchemistAPMC.

The bound from [16] is an upper bound for the dimension of the sample needed to obtain a specified confidence interval, but only when working with binomial distributions, that is, probabilities. Variance of the sample is not considered here. Let's name δ the width of our interval and α our confidence; we'll have that for any model on which we are observing a 1/0 property resulting in a probability, we won't need to run more than $N = 4 * \ln\left(\frac{2}{\alpha}\right)/\delta^2$ simulations.

The bound from [8] is computed dynamically, exploiting the increasingly meaningful knowledge about data variance that subsequent simulations give us. The principle is: less variance, smaller sample. This bound is used in AlchemistAPMC for non-binomial properties, like the mean of continuous or discrete values, but it also plays a role in keeping sample size low when working with probabilities. Basically, we gradually increase sample size and periodically stop to examine generated data: knowing how many samples we have already got and their result, and set a desired confidence, we can, for large n, compute the width of our confidence interval as

$$\delta_0 = 2t_{\alpha/2,n-1}\frac{s}{\sqrt{n}} \tag{2.1}$$

where $t_{\alpha/2,n-1}$ is the value in $\alpha/2$ of Student's t distribution with $n-1$ degrees of freedom, and s is

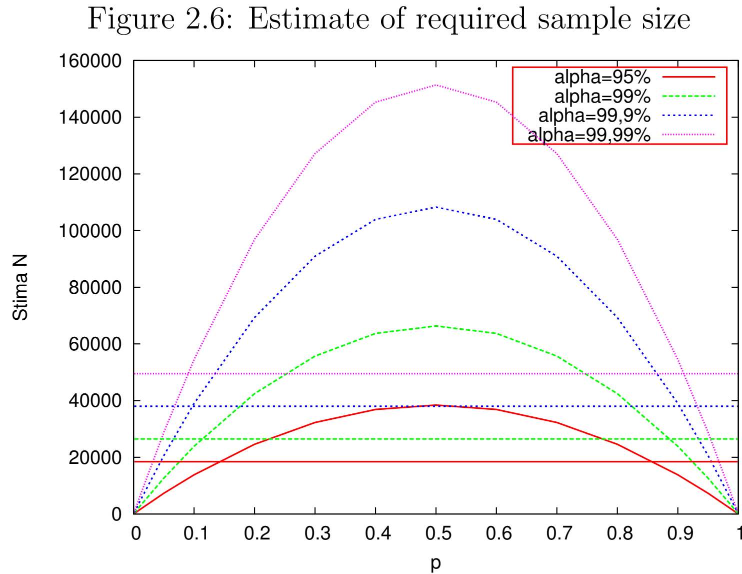
$$s^2 = \frac{\sum_{i\in[1,n]} {X_i}^2 - \overline{X}^2}{n-1} \tag{2.2}$$

Here X is the value of the observed property, which, in case of probabilities, takes value 1 for success, 0 otherwise. To obtain the desired confidence interval we run new simulations, increasing sample size, until we get $\delta_0 < \delta$.

As said before, this is the only stop criterion for non-binomial properties, but it also gives us a way better bound for binomial ones, in the case of

low variance. Figure 2.6 shows, for different values of probability (which is bound to variance in binomial distributions by the formula $s^2 = np(1-p)$)and confidence, and for $\delta = 0.01$, the value (estimated, in the case of the latter) of both of our bounds.

Figure 2.6: Estimate of required sample size



The actual necessary sample size is the minimum of the two; in practice, given we don't know probability in advance, we'll use the dynamic bound, periodically checking if we can stop, as described above, and we will stop anyway when reaching the static bound.

The dynamic method needs to be given an adequate minimum sample size, to avoid inconsistencies in cases of extremely low variance, when it is possible that the first sample contains all results of the same type, e.g. all zeros. It can be set by the user; by default it is set to the minimum number of simulations for which, in a situation of all 0s (1s), if we added a simulation with value 1 (0), the algorithm would still stop, because condition $\delta_0 < \delta$ keeps holding. This minimum is found by solving

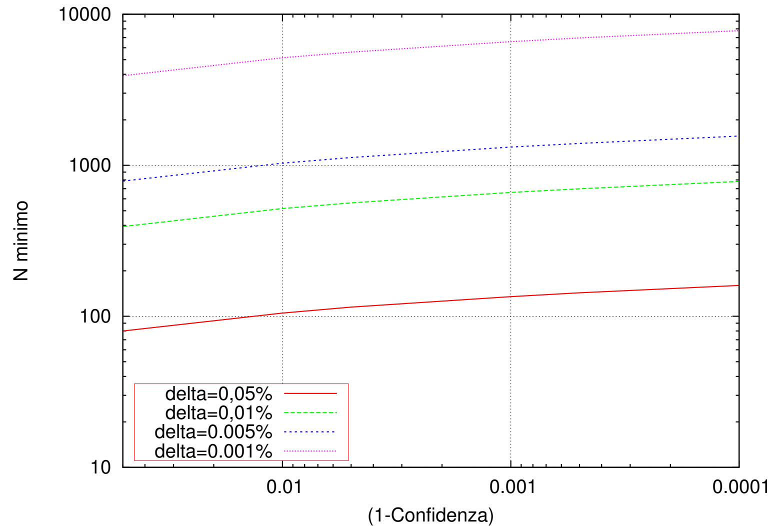
$$\delta_0 = 2t_{\alpha/2,n-1}\frac{1}{n} \tag{2.3}$$

Figure 2.7 shows how this number varies against chosen confidence interval. Variance is irrelevant here.

APMC real time preview

Statistically consistent results are available at the very end of the process, but since some experiments can take a very long time, a facility has been

Figure 2.7: Minimum sample size



developed to provide real time preview of the ongoing process for a particular class of model checking experiments, namely those experiments in which the realization of some property is checked against time. For such experiments, a chart is plotted and continuously updated, showing the probability of the observed property to be true at a certain time, along with its approximation interval. This is achieved by doing a sampling over the set of time values obtained with simulations, using a sampling interval that the user can set accordingly to the precision he requires. Other controls that can be used on the chart at any time are the setting of confidence, resulting on a different approximation interval, and the level of zoom.

Usage example

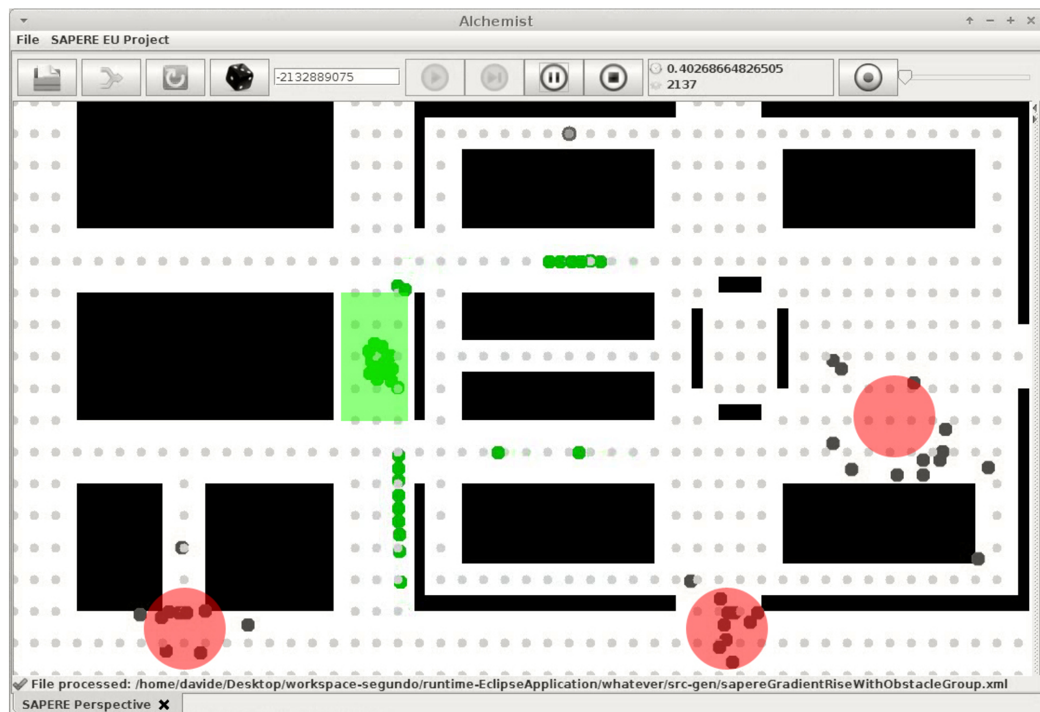
Some experiments have been made to predict with ALCHEMIST the behaviour of a group of pedestrians, exploiting the force-based model

The specification in Listing 2.11 exploits the SAPERE DSL to model an environment in which three different groups of pedestrians move to a common target. As shown in following screen shot, the environment contains some obstacles that the pedestrians, starting from the red dots, have to walk through to get to the green area, their target. In the picture, grey dots are pedestrians in their initial position, while green dots are pedestrians on their way.

Listing 2.11: SAPERE-DSL specification

```
1  environment type BikeSharingEnv
2  linking nodes in range 1.5
3
```

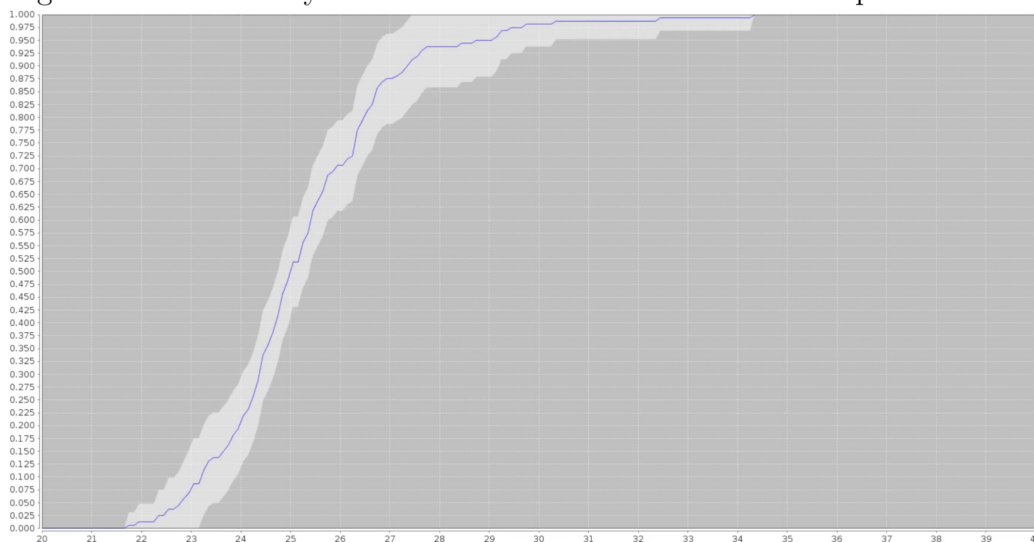
```
4   lsa source <source, Type, Distance>
5   lsa target <source, target, 0>
6   lsa gradient <grad, Type, Distance>
7   lsa crowd <crowd, L>
8
9   place 899 nodes in rect (0,0,47,18) interval 1
10  containing in point (17, 10) target
11  with reactions
12  reaction SAPEREGradient params "ENV,NODE,RANDOM,source,gradient,2,((Distance
        +#D)+(0.5*L)),crowd,2000000,10" []-->[]
13  eco-law compute_crowd []-1-> [agent CrowdSensor params "ENV,NODE"]
14
15  place 15 nodes in circle (42, 7, 5)
16  containing in all <person>
17  with reactions
18  []-100->[agent SocialForceEuropeanAgent params "ENV,NODE,RANDOM,gradient
        ,2,1,false"]
19
20  place 10 nodes in circle (33, 2, 2)
21  containing in all <person>
22  with reactions
23  []-100->[agent SocialForceEuropeanAgent params "ENV,NODE,RANDOM,gradient
        ,2,1,false"]
24
25  place 10 nodes in circle (8, 2, 3)
26  containing in all <person>
27  with reactions
28  []-100->[agent SocialForceEuropeanAgent params "ENV,NODE,RANDOM,gradient
        ,2,1,false"]
```



The experiment we perform on this model consists in observing the time required for 80% of people to reach target area. This enables us to evaluate

the effectiveness of the abstraction we choose for pedestrians. For this experiment we set a 1% approximation with a 99.9% confidence. This means that we get statistically consistent data for any probability (lowest or highest probabilities require, as stated, smaller samples) after running the simulation about 25k times. For an experiment like this, running on a commercial pc, this takes about 24 hours, but a significant preview of what is happening is available much earlier, as shown in Figures 2.8 to 2.10, obtained respectively after 160, 1k and 15k simulations.

Figure 2.8: Probability of condition satisfaction vs. time. Sample size = 160



The chart is shown as soon as a minimum sample is available, and is continuously updated, which is particularly useful in the case of complex models that take long time to simulate, giving early detection of bad behaviour, and fast previewing of the final result. To state this more clearly, let's focus on the information these three snap shot provide, and on the time required to get them on an Intel i5-3230M machine. Suppose we are trying to understand at what simulation time the probability overcomes 90%; do we really need to perform a full 25k execution? Figure 2.8, obtained in seven minutes, gives a rough representation of what is coming. The chart doesn't take much longer to get close to its final shape; Figure 2.9 is obtained after an hour, and its differences from Figure 2.10, which took fifteen hours, could be, for some uses, considered negligible. As stated, the whole model checking process would require a 25k sample, taking a whole day of running. After an hour of execution the value we are seeking is pretty well fixed in a space of two or three tenths of second between 27 and 28. Being able to get relevant data in such a shorter time is undoubtedly great advantage.

Figure 2.9: Probability of condition satisfaction vs. time. Sample size = $1k$

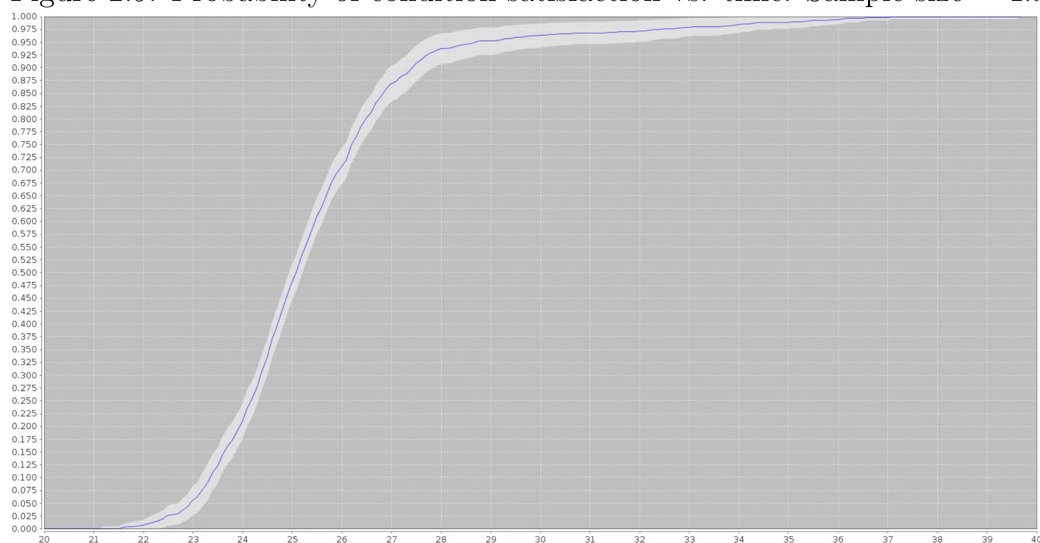


Figure 2.10: Probability of condition satisfaction vs. time. Sample size = $15k$

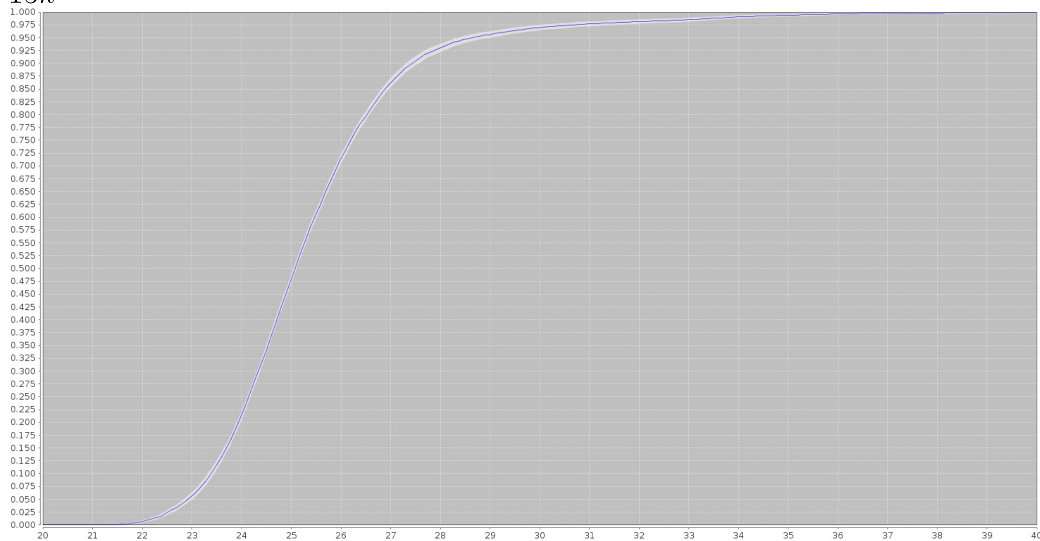


Figure 2.11:

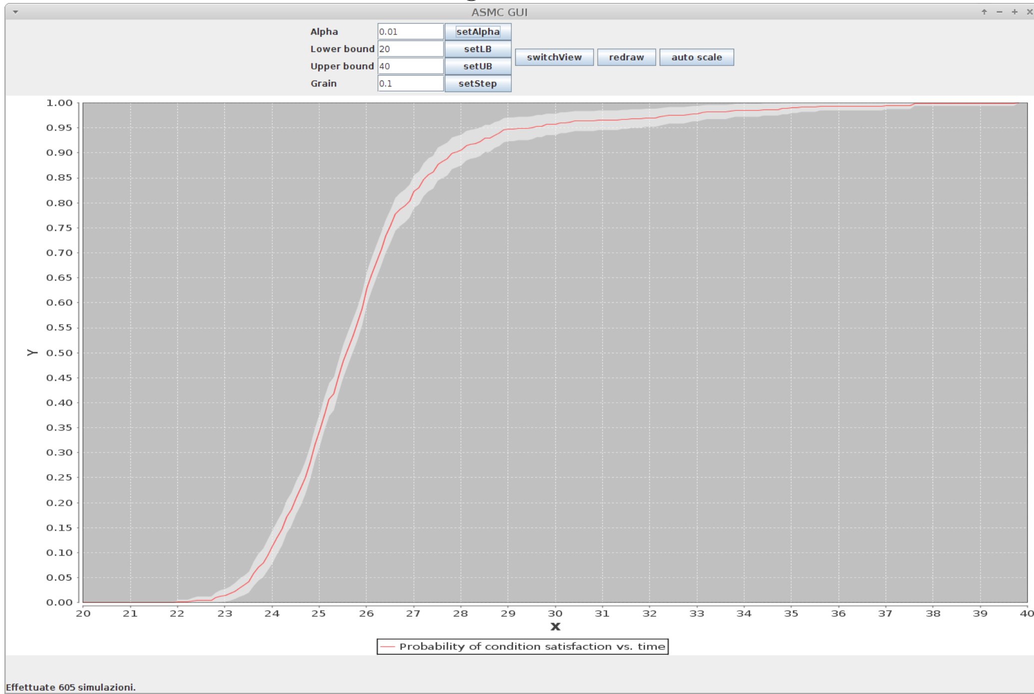


Figure 2.12:

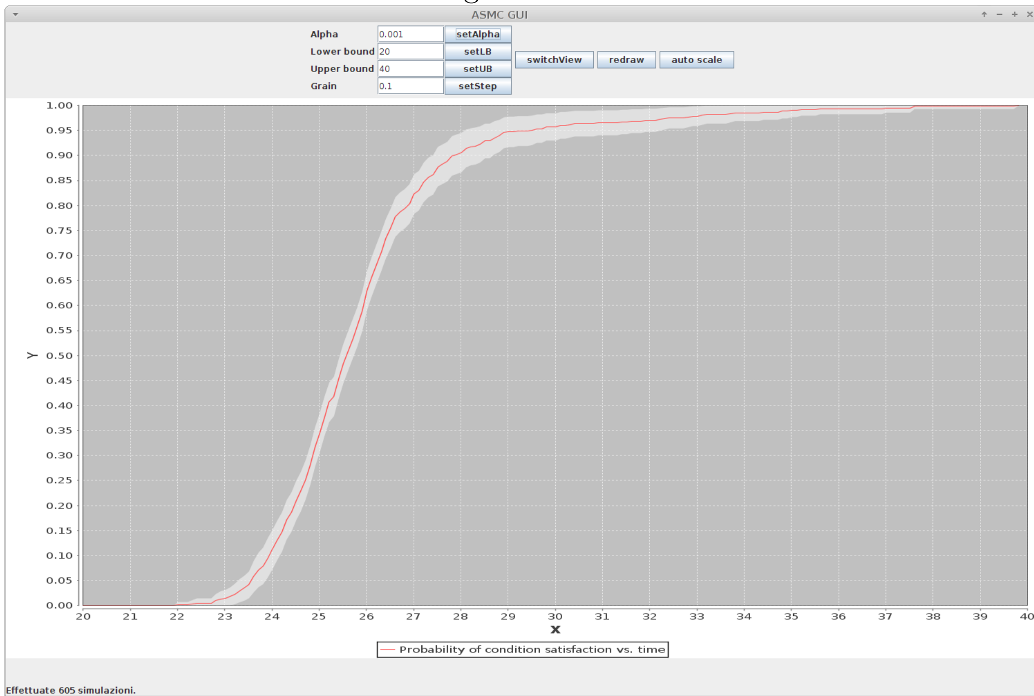
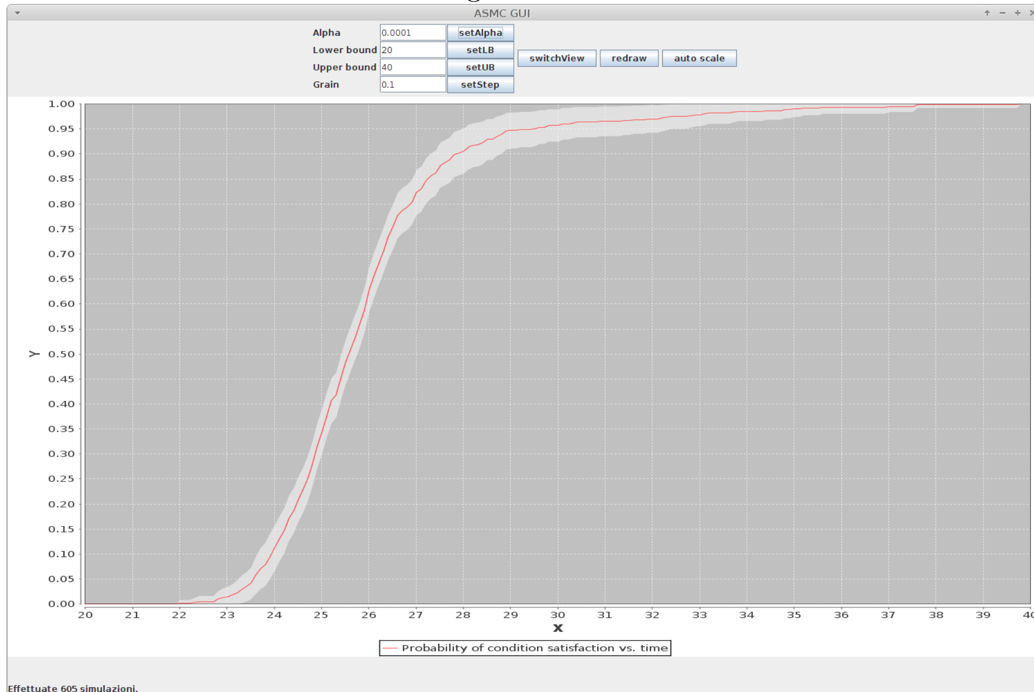


Figure 2.13:



Next three figures show how it is possible, at any time, to change desired confidence to see how it affects the approximation on the chart. User interface of the tool is also shown here. Figures 2.11 to 2.13 show three plots of the same situation, where desired confidence is set respectively to 99%, 99.9% and 99.99%.

Chapter 3

How to develop Alchemist

This chapter contains a methodology guide to ensure that developers will follow a common workflow.

3.1 Mercurial

ALCHEMIST relies on Mercurial as versioning system. Explaining the basics on this control version system is out of the scope of this manual, however a very good tutorial is available at `http://hginit.com/`. The tutorial includes a special lesson meant to re-educate those who are used to the counter-intuitive workflow of SVN. Git users, instead, should feel home when using Mercurial.

In this section, I will suppose the user understood the Mercurial basics, and I will only show how a developer could organise a good workflow, keeping the base system synchronized with the mainline and continuously integrating the new features.

Some final advices:

- The committer user name should be in the form:
  `Name Surname <youremail@yourprovider.smth>`.
- Commit **often**. A commit is a point at which you can always go back, it is cheap in terms of consumed resources and allows for much better bug hunting.
- **Never** track binaries: Mercurial is not meant to deal with those. Track only your source code, and be very careful when adding a resource (e.g. an image) in tracking.

3.2 Maven

ALCHEMIST relies on Maven to resolve the dependencies against libraries, keep them up to date, and generate the whole project documentation in a coherent and pleasant style.

3.3 PMD

From the official website description: "PMD is a source code analyzer. It finds unused variables, empty catch blocks, unnecessary object creation, and so forth". Alchemist developers must install the PMD Eclipse plugin and run it in order to be sure they produced high quality, clean code.

The PMD plugin can be installed from the update site `http://pmd.sourceforge.net/eclipse`. Once installed, the proper rules set must be set:

1. In the Eclipse properties menu, expand "PMD";
2. Select "Rules configuration";
3. Click on "Clear all";
4. Click on "Import rules set";
5. Browse your file system to find `alchemist-pmd.xml`, which contains the rules definition for Alchemist: it comes with its own rules set, meant to detect the most common badnesses not nagging the coder too much;
6. Click OK;
7. Click OK;
8. Refuse the complete build;
9. Use the freshly configured PMD plug-in by right click on your resources and Clicking PMD ⇒ Check code with PMD.

3.4 Find Bugs

Find Bugs is "a program which uses static analysis to look for bugs in Java code". Basically, its goal is to statically check for bad practices in the Java code.

As for PMD, the developers must install it and check their code, in order to be sure they have written no error prone code lines. It can be installed with the update site `http://findbugs.cs.umd.edu/eclipse/`. Once installed, it can be used by simply right-clicking on the resource of interest and running Find Bugs ⇒ Find Bugs. The code will be analysed and the problematic part (if any) will be highlighted and decorated with a fancy bug icon.

3.5 Code style

ALCHEMIST is developed by a number of people, each with its own style. This may lead to code inconsistencies and different coding styles, which, with time, may make harder for maintainers to read and fix the code. In order to prevent this situation, the developers which want to contribute to the mainline code must install Checkstyle and write their code respecting the guidelines. These guidelines are basically just the recommendations from the Java Language Specification, with some additional restriction which ensures a common code style throughout the whole codebase. All the style recommendations are compatible with the default Eclipse SDK formatting style: consequently, most of the errors can be fixed automatically by running the internal code formatter.

The Checkstyle specification to use is available at `http://alchemist-maven.apice.unibo.it/alchemist_checkstyle.xml`. For those using Eclipse, a plugin exists, it is available at `http://eclipse-cs.sourceforge.net/`. Its installation is warmly recommended.

Once the plugin has been installed, it is necessary to configure a new check configuration for ALCHEMIST: all the built-in profiles are, in fact, too restrictive. The Checkstyle options can be found under the Eclipse preferencies. In the Global Check Configurations section, click on new, and configure the new configuration as in Figure 3.1. Once done, select the Alchemist Style from the Global Check Configurations list and click "Set as Default". From now on, you can enable Checkstyle in the projects you are working on, by right click on the resource and Checkstyle ⇒ Activate Checkstyle.

3.6 Final remarks for the devels

3.6.1 Use internal logger

ALCHEMIST comes with its logger, in the class L. In order to keep the output clean, the developer should use the provided Logger for debug, warning and error throwing purposes.

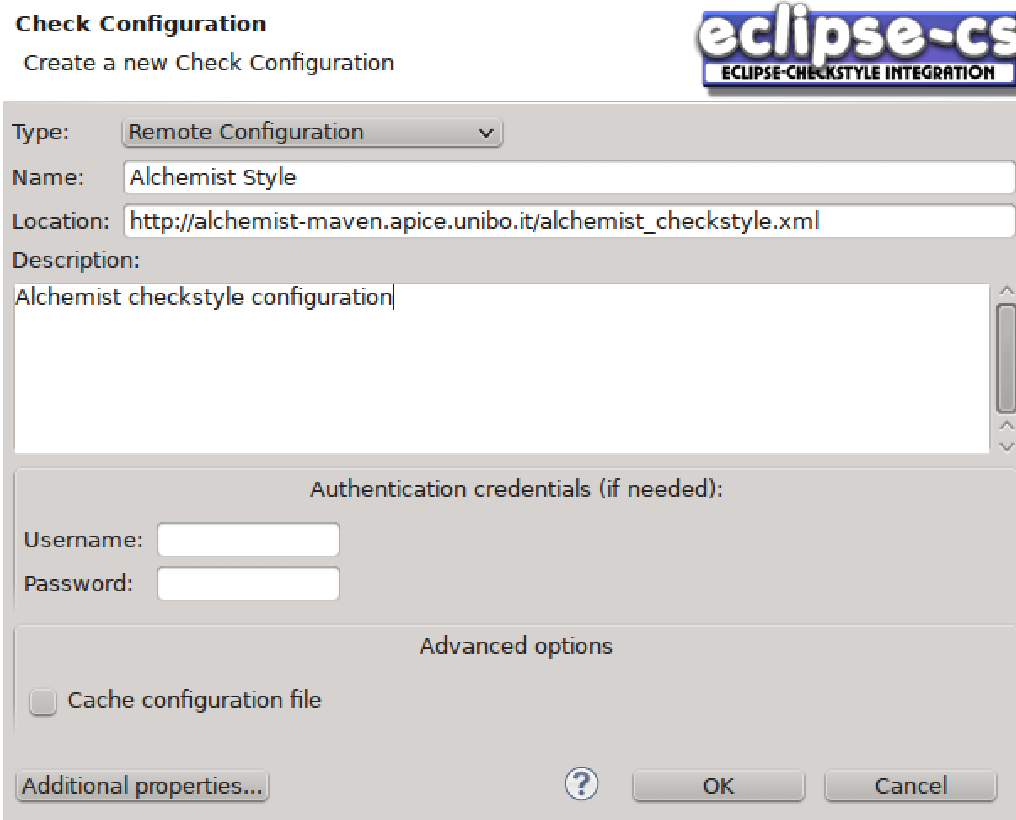


Figure 3.1: Alchemist check configuration for Eclipse-CS

3.6.2 Test plan

It is important for the developers to provide some testing of their own classes, in order to ease the whole system maintaining. ALCHEMIST is structured to provide a separate space for Java test sources and resources, in Maven style. Every time a new version is released, the Maven documentation gets regenerated, and among the other reports, the user can see the results of Surefire, which runs all the tests, and Cobertura, which analyses which and how many lines of code have actually been tested.

Currently, there is a wide portion of the code which is not tested enough, mostly because there was a single developer and the project was in embryonic stage. New contributions should not fall in the same error, and must come along with (at least partial) tests.

3.7 How to report issues

ALCHEMIST has its own issue tracking system, located on the same Bitbucket repository hosting the mainline code. It can be found at `https://bitbucket.org/danysk/alchemist/issues`. The issue tracking systems is meant to be used for bug reports, but also for enhancement requests and projects proposals.

When reporting a bug, it is important to attach everything that could help the developers to help you. In particular, I suggest reading the document at `http://www.chiark.greenend.org.uk/~sgtatham/bugs.html`, which contains a very exhaustive guide on how to correctly report issues. These are just part of the final summary:

- The first aim of a bug report is to let the programmer see the failure with their own eyes. If you can't be with them to make it fail in front of them, give them detailed instructions so that they can make it fail for themselves.

- In case the first aim doesn't succeed, and the programmer *can't* see it failing themselves, the second aim of a bug report is to describe what went wrong. Describe everything in detail. State what you saw, and also state what you expected to see. Write down the error messages, *especially* if they have numbers in.

- When your computer does something unexpected, *freeze*. Do nothing until you're calm, and don't do anything that you think might be dangerous.

- By all means try to diagnose the fault yourself if you think you can, but if you do, you should still report the symptoms as well.
- Be ready to provide extra information if the programmer needs it. If they didn't need it, they wouldn't be asking for it. They aren't being deliberately awkward.
- Write clearly. Say what you mean, and make sure it can't be misinterpreted.
- Above all, *be precise*. Programmers like precision.

Bibliography

[1] Gnu wget. http://www.gnu.org/software/wget/. Accessed: November 2, 2013.

[2] Graphhopper road routing in java with openstreetmap. http://graphhopper.com/. Accessed: November 2, 2013.

[3] Openstreetmap. http://www.openstreetmap.org/. Accessed: November 2, 2013.

[4] Osmosis - openstreetmap wiki. http://wiki.openstreetmap.org/wiki/Osmosis. Accessed: November 2, 2013.

[5] Osmosis/detailed usage 0.38 - openstreetmap wiki. http://wiki.openstreetmap.org/wiki/Osmosis/Detailed_Usage_0.38. Accessed: November 2, 2013.

[6] Planet.osm - openstreetmap wiki. http://wiki.openstreetmap.org/wiki/Planet.osm. Accessed: November 2, 2013.

[7] Xapi web service - mapquest platform. http://open.mapquestapi.com/xapi/. Accessed: November 2, 2013.

[8] G. Agha, J. Meseguer, and K. Sen. Pmaude: Rewrite-based specification language for probabilistic object systems. *Electronic Notes in Theoretical Computer Science*, 153(2):213–239, 2006.

[9] J. Berrou, J. Beecham, P. Quaglia, M. Kagarlis, and A. Gerodimos. Calibration and validation of the Legion simulation model using empirical data. In N. Waldau, P. Gattermann, H. Knoflacher, and M. Schreckenberg, editors, *Pedestrian and Evacuation Dynamics 2005*, chapter 15, pages 167–181. Springer, Berlin, Heidelberg, 2007.

[10] R. Chadha and D. Allison. Partitioning rectilinear figures into rectangles. In *Proceedings of the 1988 ACM sixteenth annual conference on*

Computer science, CSC '88, pages 102–106, New York, NY, USA, 1988. ACM.

[11] M. Chraibi, M. Freialdenhoven, A. Schadschneider, and A. Seyfried. Modeling the desired direction in a force-based model for pedestrian dynamics. 2012.

[12] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A*, 104:1876–1889, 2000.

[13] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, December 1977.

[14] D. Helbing, L. Buzna, A. Johansson, and T. Werner. Self-organized pedestrian crowd dynamics: Experiments, simulations, and design solutions. *Transportation Science*, 39(1):1–24, Feb. 2005.

[15] D. Helbing, L. Buzna, A. Johansson, and T. Werner. Self-organized pedestrian crowd dynamics: Experiments, simulations, and design solutions. *Transportation Science*, 39(1):1–24, Feb. 2005.

[16] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In B. Steffen and G. Levi, editors, *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2004.

[17] P. Lecca, A. E. C. Ihekwaba, L. Dematté, and C. Priami. Stochastic simulation of the spatio-temporal dynamics of reaction-diffusion systems: the case for the bicoid gradient. *J. Integrative Bioinformatics*, 7(1), 2010.

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

[19] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *Computer Aided Verification*, pages 266–280. Springer, 2005.

[20] A. Sharieh and W. A. Rawagepfeh. An algorithm for finding maximum indipendent set in a graph. *European Journal of Scientific Research*, 23(4):586–596, 2008.

[21] A. Slepoy, A. P. Thompson, and S. J. Plimpton. A constant-time kinetic monte carlo algorithm for simulation of large biochemical reaction networks. *The Journal of Chemical Physics*, 128(20):205101, 2008.

[22] M. Viroli, M. Casadei, S. Montagna, and F. Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Transactions on Autonomous and Adaptive Systems*, 6(2):14:1 – 14:24, June 2011.

[23] M. Viroli, E. Nardini, G. Castelli, M. Mamei, and F. Zambonelli. A coordination approach to adaptive pervasive service ecosystems. In *1st Awareness Workshop "Challenges in achieving self-awareness in autonomous systems" (AWARE 2011)*. SASO 2011, Ann Arbor, MI, USA, 7 Oct. 2011.

[24] M. Viroli and F. Zambonelli. A biochemical approach to adaptive service ecosystems. *Information Sciences*, 180(10):1876–1892, May 2010.

[25] J. Was, B. Gudowski, and P. J. Matuszyk. Social distances model of pedestrian dynamics. In S. E. Yacoubi, B. Chopard, and S. Bandini, editors, *ACRI*, volume 4173 of *Lecture Notes in Computer Science*, pages 492–501. Springer, 2006.

[26] F. Zambonelli and M. Viroli. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications*, 7(3):186–204, 2011.