Introduction to Panacea

Dario Panada
dario.panada@postgrad.manchester.ac.uk

May 3, 2016

1 Introduction

The laws of history are as absolute as the laws of physics, and if the probabilities of error are greater, it is only because history does not deal with as many humans as physics does atoms, so that individual variations count for more.
- Foundation and Empire, Isaac Asimov

Welcome to *Panacea*, a Python framework to allow the creation, simulation and observation of complex multi-agent systems and the study of emergent behaviours. This report wants to be the official framework's documentation. It should serve as a self-contained guide allowing the reader to develop familiarity with the product and enabling them to make use of it to implement their own model.

Important Notice: You are reading version *1.0* of this guide, there might be a more recent version available! This documentation refers to version *Alpha 1* of the Panacea framework. As this is not a final release, future versions might not guarantee backwards compatibility. Please check at *url* for more recent versions!

Cautionary Note: *Panacea* is currently in its *Alpha* stage. While every effort is made to provide a functioning framework, at this stage we make no guarantees with regards to stability, performance or quality.

1.1 Motivation

Complex systems are a fascinating area of study. Just looking at the natural world, we can see how some extremely complex behaviours exhibited by groups of organisms do not appear to be coordinated or controlled by any central entity. These are known as *emergent properties* of the groups, where the behaviour of single members dictated by relatively simple rules gives rise to overall group behaviours more complex than any individual could coordinate.

Examples of emergent behaviours include bird flocking, the movements of schools of fish, light emission in bioluminescent plancton and immune responses in humans. Moving away from biology, we find emergent behaviours in chemical reactions (Eg: Redox), sociology (Eg: Migration), finance (Eg: The Stock Market) and generally speaking any sufficiently complex system comprising of multiple independent agents.

Understanding such systems can allow us to obtain a better understanding of their dynamics and nature. For example, in the field of medicine, understanding the dynamics behind cancer growth and expansion can allow us to devise better, more efficient treatments. Similarly, in biology, understanding which conditions favour or inhibit proliferation of micro-organisms can be a valuable asset in research.

An alternative to *wetware* (laboratory/real world) solutions is offered by *in-silico* investigations. This approach proposes to create a virtual model of the system we are interested in allowing to simulate its evolution under various conditions and to collect useful results.

While multi-agent systems have already been studied, these often were implemented using *ad-hoc* solutions. That is, the software that was used to build and simulate such models was written *expressly* for such task and might have not suited another type of model.

Panacea wants to be a general-purpose multi-agent framework suited to implementing models studying any system or behaviour. It wishes to offer easy access to relevant tools which will allow to easily and rapidly deploy efficient simulations.

In addition, *Panacea* also wants to promote experimental reproducibility. It does this by encouraging users to define their experimental starting conditions using our own XML data structure, allowing experiments to be then validated by other users with the certainty of the setup being identical.

1.2 Presentation

As mentioned, *Panacea* is a Python framework for multi-agent systems. Key components of multi-agent systems are encapsulated in appropriate modules, where comprising classes can be instantiated and parametrized according to the user's needs. Examples include:

- **Grid** objects to represent the environment, agents can occupy space on the grids to represent their current position;
- **Coordinate** objects to represent a specific position, as well as offering useful tools to calculate distance between two positions, generating a list of neighbouring positions, etc.
- **Agent** objects to represent individual members of the simulation, as well as solutions to define their initial state and how they "evolve" at each time-step.

The aim is for users to be make use of these tools customizing them to suit the needs of their experiment. We strive to provide maximum flexibility, allowing users to redefine or extend our framework depending on their needs.

2 Architecture

2.1 The Model Object

Panacea is built using a modular approach. At the top of the model's hierarchy there is an instance of the *Model* object, or of an object inheriting from it. The Model has two main responsibilities:

- Collect all model properties, entities, agents, etc.
- Setup the initial state of the model (or delegate this task to an appropriate loader), launch the simulation with the schedule object, perform any teardown operations once the simulation has ended.

Generally speaking, once instantiated the model will its *setup*, *simulate* and *teardown* methods in this order.

2.2 Grids

Grids offer a form of spatial representation, they can be used to represent the environment. At the present moment, all grids are cartesian grids. There is a plan to include grids with hexagonal cells in one of the future releases. *Panacea* currently supports two-dimensional (2D) and three-dimensional (3D) grids. They work in exactly the same way from the point of view of the functionalities they offer, except of course a 3D grid allows an extra dimension on which to position agents.

Panacea offers two main types of grids: Numerical Grids and Object Grids. As mentioned, both of these are available as 2D or 3D grids.

2.2.1 Numerical Grids

Numerical grids hold one numerical value per position. Example uses of a numerical grid include storing environmental properties which change from position to position such as food available at a given location.

Upon instantiation, all grid positions are set to zero. Methods are available to set all grid positions to a certain value, set a particular position to a value and get the value at a certain position.

2.2.2 Object Grids

Object Grids can store one or more instance of an object, usually an agent, per position. An obvious example use is that where agents occupy a position in the environment and can move between adjacent positions.

Object Grids offer functionalities such as changing the position of an agent, finding an agent's position, retrieving all agents at a given position and finding the most/least populated position in a neighbourhood.

Agents can occupy different positions on different grids. In the case where an agent has a radius greater than one, that is when they occupy multiple grid positions in addition to the one they are centered on, the gird object will take care of ensuring the agent never accidentally "falls off the board" and other related operations.

Please note that a model does not have to be restricted to using either numerical or object grids. In fact, there can be as many grids, numerical or object, 2D or 3D, even of different sizes, as suit the user's needs.

2.3 The Schedule

In section 2.2 we discussed how spatial discretization is achieved via the use of grids. Temporal discretization, on the other hand, is achieved via the use of a *Schedule*.

The simulation's progression is divided into discrete units of time referred to as *time-steps*. At each time-step, agents are updated according to to predetermined rules. A simulation runs for a predetermined number of *epochs* or time-steps.

During the setup phase, all agents are added to the Schedule object which is then finally added to the Model object. When the Model object calls its own *simulate* method, it repeatedly calls the Schedule's main step method. At this stage, we will also mention in passing that Helper objects, a specific type of agents, are also held in the schedule and stepped at each time-step.

Each time the schedule's step method is called, the schedule shuffles the list of agents to ensure that they are never progressed in the same order (which could lead to forms of systematic bias) and then calls the each agent's step method in turn. When stepped, each agent has access to a copy of the model's instance, which represents the "state of the world". This allows agents to observe the state of grids, of other agents, to make decisions based on that and to change the state of the world/to other changes.

2.4 Coordinates

Coordinates objects are used to represent a position in a grid. Coordinates are implemented as *Coordinates2D*, representing a position in a 2D grid, and *Coordinates3D*, representing a position in a 3D grid. In addition to storing a position, Coordinate objects also offer auxiliary methods such as those to calculate the euclidean or manhattan distance between two coordinates, or to generate the coordinates representing all points in a coordinate's neighbourhood. Finally, all methods in *Panacea* to be parametrized with a coordinate will expect this to be provided as a coordinate object.

2.5 Steppables

Steppables are objects whose instances will be added to the schedule and updated at each time-step. They belong to two main classes: *Agents* and *Helpers*.

2.5.1 Agents

Agents are the main "characters" of a simulation. Each agent represents an independent entity with a *state* and a set of *rules* describing how its state is updated at each time-step.

Examples of agents could include biological cells, animals, people, etc.

Examples of state variables could include the agent's energy, its position, its age, etc.

At each time-step, the agent has access to the state of the model, which include all grids and other agents, allowing it to make a decision on how to update itself based on the current state of the simulation.

Agents should extend the *Agent* class an implement appropriate methods so that they can be integrated in the model's structure as called by the schedule. A model needs not be limited to a single type of agent, but may implement as many distinct agents as the user finds necessary.

2.5.2 Helpers

Helpers are a special class of steppables. They do not represent entities in the simulation but, rather, can be used to perform auxiliary tasks. An an analogy, they are equivalent to stagehands in theatrical productions.

Helpers are always called before agents and, contrarily to the latter, are always stepped in the same order. That is, the order in which they are added to the schedule.

Examples of uses for helpers include changing the value of numerical grids after each simulation (Eg: To simulate food resources autonomously growing/depleting), updating global model properties, removing agents to simulate natural events etc.

All helpers should extend the *Helper* class.

2.5.3 Three-Act Stepping

Panacea implements a three-act step for all steppables. The logic behind this comes from often having found ourselves in the position where we wished an agent started updating its status, then waited for all other agents to have done the same, and finally concluded its step by being able to observe the actions of all other agents.

As a practical example, consider a system of agents divided in *Donors* and *Receivers* , where the number of Donors is considerably larger than that of Receivers. Donors each start with a fixed, non-replenishable, amount of resources.

During each time-step, Donors have the goal to find a Receiver who accepts their donation, while minimizing the amount they donate. Each receiver can only accept a donation from a single donor. Receivers will select the donors who make the highest bid.

At the start, each donor will have to make a blind bid. They will then be able to look at the bids made by other contenders, and decide whether to increase, decrease or stay. Finally, Receivers will in turn select the highest bidders who, as winners of the round, see their score increased. However, as Donors' resources cannot be replenished, the more a Donor bids the less likely it is they will win a successive round. It therefore becomes a problem of balancing short-term goals with long-term goals.

We can therefore see how our three-act implementation works well allowing to effectively simulate micro-time-steps during each time-step.

In practice, each steppable implements a *stepPrologue*, *stepMain* and *stepEpilogue* method. At the start of a time-step, all steppables will see their stepPrologue method called. Once all have been stepped, the schedule will move to stepMain and finally to stepEpilogue.

It is *not mandatory* for all agents to make use of all methods. While all three must be implemented for structural consistency, nothing prevents them from simply calling a *pass* statement.

2.6 Loaders

As much as possible *Panacea* encourages users to define their model's initial parameters and setup from an external file. This enforces a separation between the model and the experiments as well as allowing users to share setup conditions among each other promoting experimental reproducibility.

Loaders are classes responsible for loading an external file describing the model's setup and modifying the model instance to attain the desired initial conditions. In general, upon instantiating the model the user will parametrize it with a string containing the setup file's path. Then, upon calling the setup method the loader will be instantiated and parametrized with the path to the setup file and the current instance of the model.

Currently, *Panacea* only supports setup files in the form of XML data-structures. The XML loader (*panacea/core/utils/SetupLoader.py*
of the *ElementTree* library to evaluate XQuery expressions against the setup file and, based on XML specifications, instantiate grid, agents, etc. An example XML document is provided in section 3.5.

2.7 Summary of Model Dynamics

We conclude this section by providing a summary of the model's dynamics and functioning.

1. The model is instantiated and optionally parametrized with the path to an external setup file;
2. The model's setup method is called. This includes parametrizing the model, instantiating all grid objects, agents, adding agents to grids where appropriate, instantiating helpers, instantiating the schedule, adding all steppables to the schedule and binding the schedule to the model;
3. The model calls the *step* method on the schedule object. The schedule then repeatedly steps all helpers and agents until a terminating condition is met. This involves repeatedly calling the stepPrologue, stepMain and stepEpilogue method on all steppables;

4. The model calls its teardown method, any end-of-execution instructions are run;
5. End of the simulation, where setup results are stored to an external file.

3 Tutorial

This tutorial is aimed at introducing the reader to the *Panacea* framework by illustrating how the framework can be used to simulate a particle swarm optimization problem. The implementation we make use of is that described at: *http://www.swarmintelligence.org/tutorials.php* You can find the full source code used in this tutorial, including XML setup files, in *panacea/examples/particleSwarmOptization*.

In this tutorial we will cover:

- Model loading and setup;
- Creating an agent class;
- Creating an helper class to visualize the state of our model;
- Customizing model setup via an external XML file.

3.1 Requirements

Panacea is optimized for Python 2.7 . In addition, the NumPy package is required.

3.2 The Basics

The first thing we will want to do when creating a new model is to extend the model class. As mentioned in section 2.1, our model instance will hold all entities and properties and be responsible for progressing the simulation. We therefore created a *PSOModelAutoSetup* class as follows:

```
class PSOModelAutoSetup(Model):

    def __init__(self, *args):
        super(PSOModelAutoSetup, self).__init__(*args)

    def setup(self):
        super(PSOModelAutoSetup, self).setup()

        self.foodLocation = Coordinates2D(self.optionalParameters["food location x"],
         self.optionalParameters["food location y"])
        self.gBestFit = -1
        self.gBest = Coordinates2D(0,0)
```

This class is relatively simple. The first thing we do in our constructor method is to pass the input parameters to the parent constructor. The parent constructor will then store the path to the external setup file and make it available to the setup method for it to call the autoloader.

Our setup method will call the parent's setup method, which is responsible for loading instantiating the autoloader and parametrize it with our setup file.

In addition, we also set the food (target) coordinates for our particles to converge to and initiate *gBestFit* to -1 and *gBest* to (0,0). *gBestFit* is inversely proportional to the distance of the closest position to the target point ever found by any particle, *gBest* is the coordinate of such position.

Take this opportunity to notice the *optionalParameters* array. When loading from an external file, we can specify model global variables. Each such parameter will have a name and a value. (Only numerical values are supported so far.) These are loaded in the optionalParameters array and can be referred to. Please note that you can only refer to them **after** you call the parent setup method. Hence, *super(*, self).setup* should always be the first instruction in your setup method.

We are now ready to instantiate our model. In a separate file (we called ours scratchpad.py) execute:

```
model = GOLModelAutoSetup("../examples/gameOfLife/xmlSetup/setupSmallModel.xml", "xml")
```

Please make sure to amend your paths so that they point to the right file. The first parameter is the location of the setup file, the second parameter is the file type. So far, *Panacea* only supports XML.

All this does so far is it instantiates the model and loads the setup files. The next step would be to add to scratchpad.py

```
model.setup()
model.simulate()
model.teardown()
```

or more simply

```
model.simulateModel()
```

which will run all three methods.

But, if we did that now, *Panacea* would throw an error as we haven't defined our agents or steppables! We will do that now.

3.3 The 'Particle' Agent

From *swarmintelligence.org*:

"*PSO is initialized with a group of random particles (solutions) and then searches for optima by updating generations. In every iteration, each particle is updated by following two "best" values. The first one is the best solution (fitness) it has achieved so far. (The fitness value is also stored.) This value is called pbest. Another "best" value that is tracked by the particle swarm optimizer is the best value, obtained so far by any particle in the population. This best value is a global best and called gbest. When a particle takes part of the population as its topological neighbors, the best value is a local best and is called lbest.*"
* In our model, local best (lbest) will be referred to as personal best(pbest).

Hence, each particle will be an instance of our agent class. As mentioned earlier, *gBest*, the global best, is tracked by the model as a global variable. *pBest*, personal best, is different for each agent.

3.3.1 Agent Setup

We will proceed to creating a *PSOAgent class*. Firstly, the constructor:

```
def __init__(self, *args):
       if(len(args) == 1):
           args = args[0]

       radius = int(args[0])

       super(PSOAgent, self).__init__(radius)

       # Each particle will have a random color.
       self.displayColor = "#%06x" % randint(0, 0xFFFFFF)

       self.pBestFit = 0
       self.pBest = Coordinates2D(0,0)

       self.c1 = self.c2 = 2
```

The first few lines are simply to account for the fact parameters can be provided as a list object (Eg: [param1, param2, param3]), which is the case when external setup is used, or individually. We then proceed to calling the parent constructor passing the agent's radius (in this case 1, that is, each particle occupies a single grid cell).

We then give each particle a random display color that will be used by our renderer and set *pBestFit* to 0 and *pBest* to (0,0). Finally, *c1* and *c2* are two constants which we will use later on to update the agent's position and which are generally set to 2. We have now defined what will happen at agent instantiation, we now need to define how the agent will update itself.

3.3.2 Agent Stepping

We now proceed to defining the *stepPrologue*, *stepMain* and *stepEpilogue* methods.

stepPrologue

In the stepPrologue method we calculate the fitness of the current agent's position by looking at its current distance from the target. If the current position is closer than any other position, then we store it as *pBest*.

```
def stepPrologue(self, model):
        grid = model.getGridFromName("psogrid")

        # Fitness is inversely proportional to the distance from the food Location
        self.pCurrent = grid.getAgentPosition(self)

        dist = grid.getAgentPosition(self).getEuclideanDistanceFromCoordinate(model.foodLocation)

        # Preventing division by zero
        if(dist <= 0):
            self.pCurrentFit = 1000000
        else:
            self.pCurrentFit = 1/dist

        # Looks like we've done our best yet
        if(self.pCurrentFit > self.pBestFit):
            self.pBestFit = self.pCurrentFit
            self.pBest = self.pCurrent
```

In the first instance, we retrive our grid from the model using the *getGridFromName* method and passing the name of our grid as a parameter *psogrid*. We then use the *getAgentPosition* Grid method to the get the coordinates of the current agent.

* This example was written at a *pre-Alpha* stage. In *Alpha 1*, the current version, agent grid positions are also stored in the agent instance by accessing the array *gridPositions*. So, as an alternative to the first two lines, we could use:

```
self.pCurrent = self.gridPositions["psogrid"]
```

In both cases, *dist* will be a *Coordinates2D* object referring to the particle's current position.

Hence, we can use the *getEuclideanDistance* method to calculate the distance between the current position and the target position. (The latter is stored as a global parameter of the model and can be accessed as *model.foodLocation*.

pCurrentFit, that is the fitness of the current position, is inversely proportional to the distance. If the distance is zero, we simply set CurrentFit to a very high value, otherwise it will be $\frac{1}{dist}$.

Finally, if the current fit is greater than the previous best fit, that is if we are closer to the target than we have ever been, we store the current fitness and position as new personal bests.

stepMain

The stepMain method is concerned with determining whether, at the current time-step, any agent has found a position closer to the target than any agent ever had before.

```
def stepMain(self, model):
        if(self.pCurrentFit > model.gBestFit):
            model.gBestFit = self.pCurrentFit
            model.gBest = self.pCurrent
```

If that is the case, we replace *gBestFit* with the agent's *pCurrentFit* and the model's *gBest* with the agent's *pCurrent*.

stepEpilogue

Finally, stepEpilogue is concerned with actually *moving* agents.

```
def stepEpilogue(self, model):

        # Standard updating according to PSO rules. And taking
        # advantage of our custom sum and mul operators
        # in the coordinate objects
        movement = self.c1*random()*(self.pBest-self.pCurrent) +
        self.c2*random()*(model.gBest-self.pCurrent)

        g = model.getGridFromName("psogrid")

        # In light of recent changes, object grid positions are now
        # also stored in the agent itself as well as in the
        # grid object, so this could be changed.
        pos = g.getAgentPosition(self)


        g.moveAgent(pos+movement, self)
```

The displacement, stored in the *movement* variables, represents the vector we are adding to the agent's current position and is dependent on *pBest*, *pCurrent* and *gBest*. Note that because coordinates have to be whole numbers results of coordinate arithmetic are casted to integers from within the various Coordinates objects.

Finally, we call the *moveAgent* method on the grid object parametrizing it with the agent we want to move and the position we want to move it to.

3.4 The Grid Display Helper

Although our model is now fully ready, we also want to be able to follow the movement of our particles in real-time. Because of this, we will extend *Panacea*'s *Grid Display Helper* to provide a *ParticleSwarmOptimizationRenderer* class. This instance will be stepped once per time-step and will be responsible for updating a canvas re-drawing the position of each particle in their corresponding position. Remember that, just as agents, helpers are added to the schedule and have their stepPrologue, stepMain and stepEpilogue methods called once per timestep. They are therefore ideally suited to tasks such as updating canvases to reflect changes in the model.

```
    def __init__(self,*args):

        # Not too pretty but obviously we can't load an object from an xml file,
        # so we'll be a bit creative about this just for this example and check whether
        # our grid variable is a grid object or a string pointing to the grid

        grid = args[0]

        if(isinstance(grid, ObjectGrid2D)):
            self.setupGrid(grid)
            self.isSetup = True
        else:
            self.gridName = grid
            self.isSetup = False
```

We start with the constructor method, where we want to specify the grid that we will be rendering. We make allowance for the grid to be passed both as an actual grid object or as a string representing the grid's name.

```
class ParticleSwarmOptimizationRenderer(GridDisplayHelper):

    def stepPrologue(self, model):
        if(self.isSetup == False):
            grid = model.getGridFromName("psogrid")
            self.setupGrid(grid)
            self.isSetup = True
    def setupGrid(self, grid):
        self.top = Tkinter.Tk()

        self.multiplier = 5

        self.gridSize = grid.getSize()
        self.gridArray = grid.getGrid()



        self.C = Tkinter.Canvas(self.top, bg="black", height = self.gridSize[1]*self.multiplier,
                   width=self.gridSize[0]*self.multiplier)
```

We make use of our prologue to check if we have retrieved the grid object to render. If not, we retrieve it from the model and draw the initial canvas.
* At this stage we are using the Tkinter library for demonstration purposes.

The updating of the canvas can take place either in stepMain or stepEpilogue. In this case we have opted for stepEpilogue, hence:

```
def stepEpilogue(self, model):
        self.renderGrid(model.getGridFromName("psogrid"), model.foodLocation)

    def stepMain(self, model):
        pass
```

Each time stepEpilogue is called, we will be calling our renderGrid method passing the current state of the grid as parameter and the location of the target.

Finally, the *renderGrid* method simply updates the canvas by erasing everything that was on it previously, looping through agent and drawing a circle at the canvas position corresponding to its coordinates and drawing a square on the target point.

```
def renderGrid(self, grid, foodLocation):

        self.C.create_rectangle(0,0,self.gridSize[1]*self.multiplier,self.gridSize[0]*self.multiplier,
            fill="black")

        for x in range(0,self.gridSize[1]):
            for y in range(0,self.gridSize[0]):

                agent = grid.getAtPos(Coordinates2D(x,y))

                if(len(agent) == 0):
                    continue

                agent = agent[0]

                mx = x*self.multiplier
                my = y*self.multiplier

                p = self.C.create_oval(mx,my,mx+self.multiplier, my+self.multiplier, fill=agent.displayColor)

        foodLocation = foodLocation.getCoordinates()
        p = self.C.create_rectangle(foodLocation[0]*self.multiplier,
        foodLocation[1]*self.multiplier,
        foodLocation[0]*self.multiplier
        +self.multiplier,
        foodLocation[1]*self.multiplier
        +self.multiplier, fill="red")
        self.C.update()
        self.C.pack()
        sleep(0.2)
```

3.5 External Setup

Once all classes have been created (Model, Agents and Helpers) it is possible to define the initial setup of the model in an external XML file. This brings the advantage of being able to separate our code from our model's structure, changing initial conditions without having to touch any of our code as well as easily sharing experimental setups with colleagues. Below, our experimental setup. (Only one particle is instantiated, more can be instantiated by adding XML agent nodes.)

```
<model name="particle swarm optimization example" epochs="100">

    <modelParameters>

        <parameter name="food location x" value="50" />
        <parameter name="food location y" value="60" />

    </modelParameters>

    <grids>

        <gridIndex>

            <grid ref="ObjectGrid2D" module="panacea.core.Grid" class="ObjectGrid2D"/>

        </gridIndex>

        <modelGrids>

            <grid type="ObjectGrid2D">

                <parameters>
                    <parameter name="xsize" value="100"/>
                    <parameter name="ysize" value="100"/>
                    <parmater name="name" value="psogrid"/>
                </parameters>

            </grid>

        </modelGrids>

    </grids>

    <helpers>

        <helperIndex>

            <helper ref="PSOR" module="panacea.examples.particleSwarmOptimization.ParticleSwarmOptimization"
                    class="ParticleSwarmOptimizationRenderer"/>

        </helperIndex>

        <modelHelpers>

            <helper type="PSOR">

                <parameters>
                </parameters>

            </helper>

        </modelHelpers>

    </helpers>

    <agents>

        <agentIndex>

            <agent ref="PSOAgent" module="panacea.examples.particleSwarmOptimization.ParticleSwarmOptimization

        </agentIndex>

        <modelAgents>

            <agent type="PSOAgent">

                <parameters>

                    <paramter name="radius" value="1"/>

                </parameters>

                <gridPositions>

                    <gridPosition grid="psogrid">
                        <coordinate name="x" value="10"/>
                        <coordinate name="y" value="10"/>
                    </gridPosition>

                </gridPositions>

            </agent>

            <agent type="PSOAgent">

                <parameters>

                    <paramter name="radius" value="1"/>

                </parameters>

                <gridPositions>

                    <gridPosition grid="psogrid">
                        <coordinate name="x" value="20"/>
                        <coordinate name="y" value="20"/>
                    </gridPosition>

                </gridPositions>

            </agent>


        </modelAgents>
        </agents>

    </model>
```

The above XML can be adapted to include additional gird classes and grid instances. Similarly, additional agent classes and agent instances can be added by similar means.

The user can simply add the pointers to the new agent/grid classes in gridIndex/agentIndex and then instantiate them as children of modelAgents or modelGrids. Obviously, the corresponding class files will have to extend *Panacea*'s objects and implement appropriate methods.

A *schematron* XML schema file is provided so you can validate your XML setup and ensure it is compliant with what the framework expects.

4 Extending the Model

As we hope to have illustrated in this manual, *Panacea* is designed to offer a series of tools to facilitate the developers' job while allowing maximum flexibility in how these are extended are implemented. All classes Coordinates, Grids and Steppables (Agents and Helpers) extend their parent classes, which provide useful functions to handle setup and instantiation while describing the skeleton these objects should adopt to integrate in the model.

Should users wish to implement their own custom girds or coordinates they can do so by extending the appropriate parent classes. The same is for all steppables. All new objects can then be loaded via the external XML setup process by referencing them in the *gridIndex*, *agentIndex* or *helperIndex* nodes.

5 License

MIT License

Copyright (c) 2016 Dario Panada

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.