The DEECo Playground Framework: Scenario Creation Guide

The document serves as a guide to the creation of scenarios with the framework. The process of scenario creation is illustrated on two examples.

The first example scenario is very simple. It shows the basic elements of which scenarios are composed, and what has to be done to set a scenario up.

The second example shows a complex scenario. It demonstrates how the environment can be extended with an additional complex layer, and how to make robots interact with this layer.

Both the described scenarios, along with several others, can be found in the project's repository.

Contents

---

1 Creating a simple scenario

The scenario features two types of robots. There is a one robot of the type `PredatorRobot`, which moves randomly through the field. All the other robots are of the type `PreyRobot`, and are programmed to run away from the predator, if it appears near them. The prey robots receive the information about the position of the predator through an ensemble.

In the creation of scenarios, it is useful to divide a system being designed into two parts: a physical part, and a computational part. A physical part is defined by layers of the environment, and the implementation of robots' wheels. A computational part is represented with classes of components and ensembles. The physical part defines the context in which robots operate: the

world they live in, and their capabilities. The computational part is basically the programs that control the behavior of robots in that environment. Before those programs could be written, it has to be known in which context the robots will operate. Therefore, the physical part has to be defined first.

1.1 Adding an additional environment layer

By default, robots do not know their positions in the environment. They receive only the information about collisions through the default sensor, but in most cases this information alone is not sufficient to make any meaningful decisions. In order to make their actions more intelligent, the robots need to be provided with an orientation system. The simplest way to do this is to give robots a sensor that will provide them with information about their current coordinates.

This can be achieved by defining a new environment layer. Each class that represents environment layer has to extend `SensoryInputsProcessor` (`SIP`) abstract class parametrized by the type of inputs it generates for corresponding sensors. Since this layer provides robots with information about their coordinates, its input type is `Coordinates`. The implementation of this layer is shown in the listing below.

```
public class CoordinatesProcessor
        extends SensoryInputsProcessor<Coordinates> {

    @Override
    protected List<Coordinates> sendInputs(
            List<RobotPlacement> robots,
            List<ObjectPlacement> objects
    ) {
        List<Coordinates> coordinates = new ArrayList<>();
        for (RobotPlacement r : robots) {
            coordinates.add(new Coordinates(r.getX(),
                    r.getY(), r.getAngle()));
        }
        return coordinates;
    }
}
```

In the method `sendInputs`, `Environment` provides the layer with lists of robots and objects, through which it can access various parameters of these entities. In this example, the methods `getX`, `getY`, and `getAngle` are used to construct the robots' coordinates. Thus, the robots with the sensor of this type will use the same coordinate system as does the simulation. It has to be noted, that the list that an `SIP` returns has to contain inputs for every robot, in the same order in which those robots appear in the `robots` list. If the returned list will have incorrect size, the simulation will exit with exception.

This is the simplest possible example of an additional environment layer, since it does not compute or generate any data on its own. It only transmits the coordinates from the environment to robots.

To ensure that robots have access to these data, two things have to be done. First, the created `SIP` has to be added to the scenario file. The following XML node would correspond to this layer:

```
<sensor name="coordinates" processor="package.CoordinatesProcessor"/>
```

Second, the sensor corresponding to this layer has to be registered on each robot. To do this, the following line has to be added in the robot's constructor, or in the `processArg` initialization method.

```
this.sensor.registerSensor("coordinates");
```

Then, a robot can access the data from its sensors in the following way (where the `sensor` variable is a robot's `SensorySystem`):

```
CollisionData collisionData = sensor.getInputFromSensor("collisions",
    CollisionData.class);
Coordinates coordinates = sensor.getInputFromSensor("coordinates", Coordinates.
    class);
```

1.2 Programming wheels

The `Wheels` interface, which represents robots' actuators, has two methods. The first one is `setAction`, which is called by robot when it chooses its next action. In this scenario, only the simplest implementation of this method is needed. It is shown in the listing below.

```
public void setAction(double speed, double angle) {
    this.speed = speed;
    this.rotationAngle = angle;
}
```

The second method is called by the environment in each cycle to receive a robot's action. In this scenario, robots cannot move forward and rotate at the same time. This is achieved by the following implementation of the method:

```
public Action sendCurrentAction(int cycle) {
    if (this.speed != 0) {
        this.rotationAngle = 0;
    }
    return new Action(this.speed, this.rotationAngle);
}
```

Thus, if robot tries to move forward and rotate at the same time, only the forward movement is executed.

1.3 Programming robots and ensembles

Now, the robots can access all the necessary information through the sensors, and can move through the field with their wheels. The remaining part, for this scenario, is simple.

Each robot has a single process that contains its decision making routine. In this process, robot examines the data available to it, and decides which action to take based on these data. The data available to robot can be received either through sensors or through an ensemble. When the `PredatorRobot` appears near a `PreyRobot`, the `PreyRobot` receives an alert signal through the `SignalEnsemble`. With this signal, it receives the coordinates of the predator. From the received coordinates and its own coordinates, the robot calculates the optimal direction to run away from the predator, and commands its wheels to move in that direction.

2 Creating a complex scenario

The second scenario features firefighter robots that have an objective to extinguish fires that appear randomly on the field. There are five firefighter robots in the scenario. One of them is the leader of the firefighter team, that can give orders to others. These robots are equipped with fire extinguishers that give them ability to reduce temperature around them and extinguish fires. All movements of the robots cost them energy, and if robot runs out of energy, it breaks and is not able to move anymore. Robots can recharge their batteries using a special charger station object placed in the center of the the field.

The environment itself contains an additional temperature layer, in which fires appear. This layer determines local temperature at every point of the field, and generates data for temperature sensors, which each firefighter has. When robot operates in high temperature, its energy gets depleted faster. For firefighters' team leader, there is also an ability to get information about positions of fires appearing on the screen, so that robots are not limited with information about the temperature at their current location.

To coordinate their actions, robots are provided with the knowledge of their current positions.

2.1 Layers of the environment

The scenario has two additional layers of the environment. The first one is the coordinates layer, which is represented with the same `CoordinatesProcessor` class shown in the previous example.

The second one is the `EnergyTemperatureProcessor`, responsible for managing robots' energies, and temperatures of the field. Since robots should not be able to change their own energy levels directly, their energies are managed externally, by the environment. And since high temperatures should affect these energy levels, the energies and the temperatures are put in the same layer.

Thus, this layer has two mainly independent responsibilities. The first one is to hold the map of temperatures of the field, and to calculate how it evolves through time, accounting for the impact of fires appearing on the field and active fire extinguishers. The map of temperatures is represented as an array of temperature values of each pixel-unit of the field. Each cycle, this layer calculates the effects of heat exchange between the adjacent pixels. With some probability, it also generates a new fire at a random point on the field.

The second responsibility of the layer is to manage energies of the robots. The robots' energies are affected by several different factors. First, a robot's energy is reduced by the actions taken: by movements and activation of fire extinguishers. Second, the energy is reduced even more, if a robot operates in the high temperature. Third, the energy gets restored if a robot is located near the charging station.

The input generated by this layer for robots' sensors contains the following information: an amount of energy a robot has at the moment, the damage it currently receives from high temperature, a maximum temperature that was detected in the area covered by the robot's body, and a temperature vector that represents an angle from which the maximum temperature comes from. For the leader of the team, this input also contains a list of coordinates of fires on the field.

This input provides robots with all the information they need to perform their task.

To calculate the temperatures correctly, the `EnergyTemperatureProcessor` needs to know the positions of the fire extinguishers on the field. And since a fire extinguisher can be turned on and off, it is not enough to know the robots' positions. This means that robots need to have an ability to act upon the temperature layer of the environment. Thus, their wheels have to be programmed differently.

2.2 Firefighters' wheels

The wheels of the firefighter robots have to satisfy two requirements. First, they have to work only if a robot has enough energy. Second, they have to be able to act not only upon the basic layer of environment, but also upon the temperature layer.

The first requirement is satisfied by defining a new method `provideEnergy` on the wheels, which takes a reference to a robot's `SensorySystem`. If it finds out that robot's energy sensor shows that it does not have enough energy, it does not permit wheels to send an action that robot wants. Robot has to call this method every time it wants to move. If this method will not be called, or if the robot does not have enough energy, the wheels will not send the action robot wants to perform, and the robot will not move. It is not possible for robot to falsify the data it receives from sensors. This way, an additional physical constraint is implemented in the scenario, without affecting the code of the robot classes.

The second requirement is satisfied by defining a new subclass of the class `Action`, the `EnergyTemperatureAction`. This subclass contains a boolean field `isExtinguisherActivated`

that determines the state of a robot's fire extinguisher. The implementation of wheels is also extended with methods `activateExtinguisher` and `deactivateExtinguisher` to give robots an ability to control their fire extinguishers.

The `EnergyTemperatureProcessor` accesses current actions of robots from the list of `RobotPlacements` that it receives in the `sendInputs` method. If an action is an instance of the `EnergyTemperatureAction`, the `EnergyTemperatureProcessor` casts it to that type to access its additional fields. This way, robots can impact additional layers of the environment.

The wheels of the leader of the firefighters' team are extended by an additional capability. The leader needs to know the positions of all the fires on the field. It receives this information by sending `extendedDataRequest` in its `Action` to the environment. In response, the `EnergyTemperatureProcessor` adds the list of fire positions to the leader's sensory input. This request costs leader a lot of energy, so the leader's wheels will not send it if the leader is in danger.

2.3 Behavior of firefighter robots

Firefighter robots can operate in two modes: controlled mode and autonomous mode. Usually robots operate in controlled mode, in which they execute orders of the team leader. When the leader gets broken due to low energy (and thus cannot receive information about positions of fires on the field anymore), the remaining robots switch to autonomous mode.

The leader collects sensory data from all the firefighters through the `DataAggregationAndOrderDistribution` ensemble. Based on these data and its own information about fires on the field, it calculates optimal destinations for each firefighter in the team. The leader's orders are distributed through the same ensemble.

In autonomous mode, robots walk through the field, searching for fires. When a robot detects an unusually high temperature, it follows the temperature vector until it finds a fire. Then, the robot activates its fire extinguisher to get rid of the fire. When it detects low energy on its sensor, it returns to the charging station. Sometimes, usually when a road to the station lies through a field of fires, robot gets broken on its way.

The charging station is a simple object without any processes. It informs robots about its position through an ensemble. The switch between the two operating modes is regulated by another ensemble.

2.4 Visualization settings

A scenario can always be visualized with the default configuration. In the case of this scenario, however, the resulting animation will not give viewers understanding of what is happening on the screen. Figure 1 shows the firefighters scenario visualized with the default configuration.

Appropriate visualization settings for this scenario have to reflect the states of the robots (following an order, driving to the charging station, broken, etc.), and to display the distribution of temperatures on the field. For this reason, this scenario has to be visualized with a custom configuration file.

The configuration file written for this scenario satisfies the first requirement using the tag-based visualization of the robots layer. In the process of simulation robots change their tag strings according to the states they are in. If for example, a robot has low energy, and rides to the charging station, it sets its tag to `"RECHARGE"`. Then, during the visualization, those tags are extracted from the simulation logs. The configuration file specifies textures corresponding to the specific tags. Based on these settings, the visualization program chooses different textures for robots. Thus, the textures of robots can change from cycle to cycle.

The temperature layer of the environment is visualized using an additional visualization layer. It is represented with the `TemperatureLayer` class. In each simulation cycle, `EnergyTemperatureProcessor` writes the state of the temperature layer into a file. A path to this

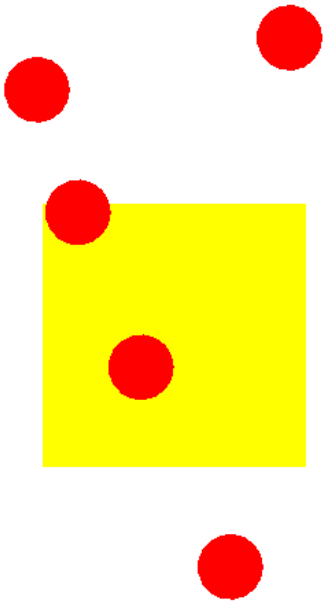

Figure 1: Firefighters scenario with the default visualization configuration.

file is provided to it in the scenario file. Then, the same path is provided to the `TemperatureLayer` in configuration file. From this file, the `TemperatureLayer` reconstructs a temperature map for each cycle, and draws it on the screen.

The remaining settings are simple. There are three layers in this visualization. The additional temperature layer is drawn first. On top of it, the texture representing the charging station object is drawn. The robot layer is rendered last. The configuration file also includes the specification of some global parameters, like zoom and speed of the visualization.

Figures 2 and 3 show the resulting picture.

3 Using other features of the application

Besides the two described scenarios there are several others that can be found in the project's repository. Some of those scenarios use bitmaps of obstacles, which are not used in the two described scenarios. It is not hard to include a bitmap in a scenario, and the functionality of bitmaps was described in the Implementation and Architecture document.

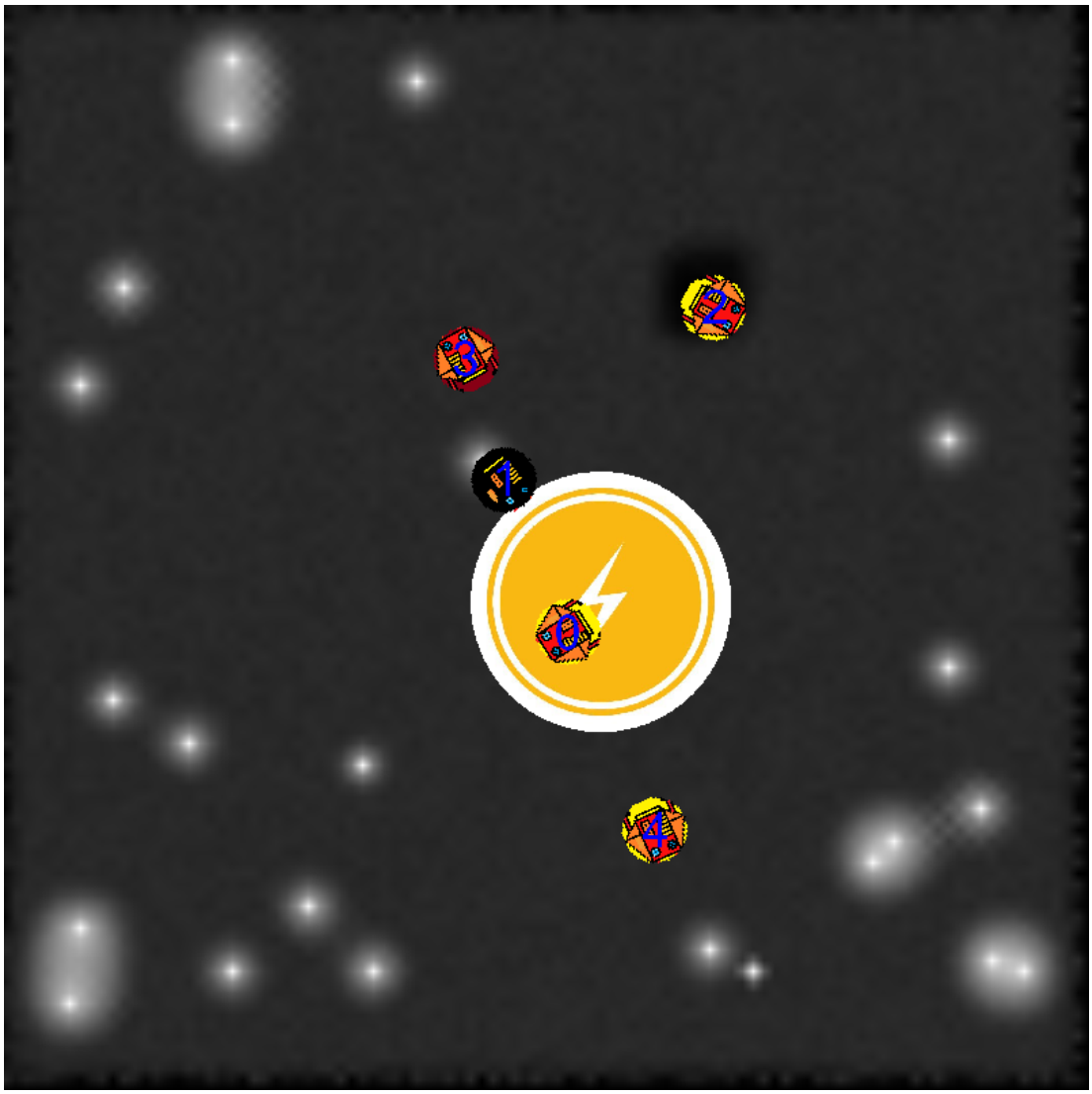

Figure 2: Firefighters scenario visualized with a configuration file written specifically for it.

The Competition scenario, pictured on the Figure 4 of that document uses some other features of the application. It has two teams of robots competing for items that appear on the field. It uses a special field of the `Coordinator`, the `status` field, to display the information about the current score. Another special field, `endSignal`, is used to finish the simulation when a team receives enough points. The items that these robots collect are small objects that are programmed to disappear, when a robot shows up near them, and to reappear later at a random location (a team receives a point for each item collected in this way). The other object calculates the score of the game, and updates the status.

The playground does not set any limits on the number of robots, objects, ensembles, and additional layers of the environment that user can add to the scenario. However, with the growing number of the DEECo components, and especially ensembles, the duration of simulation increases. The main performance bottleneck for the application is the number of DEECo tasks executed in each cycle of the simulation. And since a deployment of an ensemble creates a separate task for each deployed component, adding an ensemble is usually much more costly in terms of the increased duration of the simulation, then adding a component.

Figure 3: Firefighters scenario: robots #1 and #2 are broken; robot #3 is in danger and returns to the charging station.