The DEECo Playground Framework: Implementation and Architecture

Contents

1 Structure of the framework

The framework is divided into two separate programs: Simulation and Visualization. Those programs are executed independently, and the only thing that connects them to a single framework is the format of simulation logs files. These files store the results of the simulation, so that later they could be read and visualized by the Visualization program. Those two programs, along with the format of the simulation logs, are described in their corresponding sections below.

Figure 1 shows a general scheme of interaction between the user and the framework.

The only input that the Simulation program receives from a user is an XML file with a scenario description. This file contains references to other resources used in the scenario: classes of entities presented in the scenario and, optionally, a path to a bitmap with the description of physical obstacles. The program can receive several scenario files at once, in this case the provided scenarios will be simulated one by one.

For the Visualization, the main input is the simulation logs file generated by the Simulation program. There can be also an optional second input: an XML file with visualization configuration. In the same way as a scenario description file, a configuration file can also contain references to other resources, mostly to images and Java classes.

Both scenario description files and configuration files have corresponding XML Schemas: `Simulation.xsd` and `Visualization.xsd`. The files provided by user should be valid against those schemas. The formats of those files are described in detail in user manual available in the project's repository.

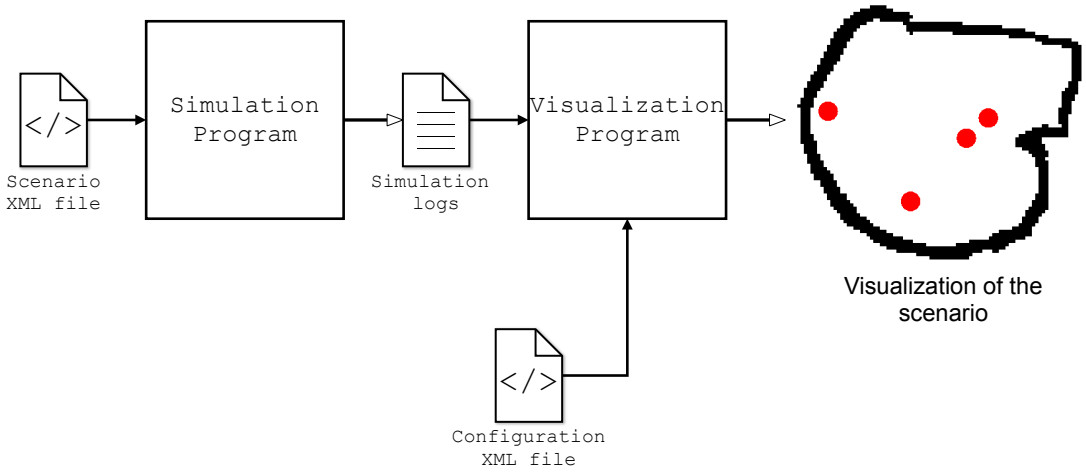

Figure 1: Inputs and outputs of the playground.

2 Simulation

The playground is built on top of the JDEECo runtime. It connects this runtime to a virtual simulation of the external physical environment. The components living in JDEECo Runtime can communicate with the environment only through their sensors and actuators. So the main challenge here was to design the environment itself and to establish proper connections between the runtime and the environment. Thus, logically the simulation can be divided into two main interconnected subsystems: the JDEECo Runtime, and the environment. This division is shown on the figure 2

Both subsystems are initialized and instantiated by the `Simulation` class. There is a special component, the `Coordinator`, that interconnects the two subsystems. This component also ensures a correct flow of the simulation, dividing it into cycles. In the following subsections the two subsystems are examined separately. Then, the implementation of their connections is explained.

2.1 Environment subsystem

The `Environment` class has a single static instance that can be accessed globally, from the different components and ensembles inside of the JDEECo runtime. This class provides access to all environment-related functionality, and thus combines several responsibilities.

The main purpose of the environment is to represent a physical world in which robots exist. As the actual physical world is very complex, this representation imitates only its basic features. The world responds to robots' actions by moving them through the field and resolving collisions that occur between them. To robots, the external world is also a source of information they receive through their sensors. For each type of sensor, there exists a layer of the environment responsible for generating data for it.

Each layer of the environment is represented with an instance of the `SensoryInputProcessor` class (`SIP`). By default, there is only one basic layer of environment in the simulation. Users can add additional layers by defining new implementations of `SensoryInputProcessor` class and adding corresponding nodes to a scenario file. An example of the creation of additional layers of the environment is shown in the scenario creation guide. Each `SIP` class is parametrized by the type of input it generates for robots' sensors. It has the method `sendInputs` that returns a list of inputs for the sensor that corresponds to this layer. This method is called in every simulation cycle by the `Environment`. `SIP`s can access the lists of robots and objects and the

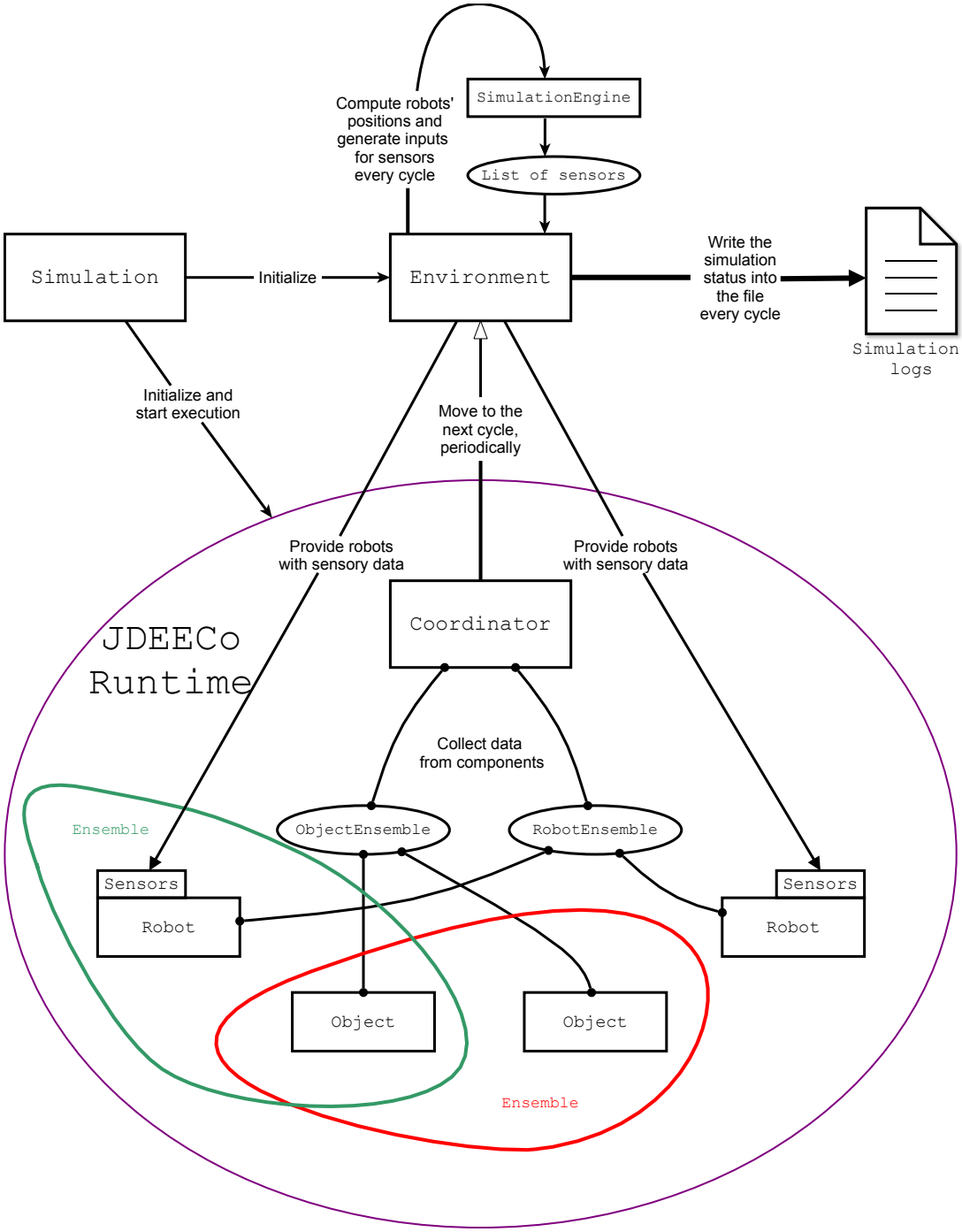

Figure 2: The general scheme of the simulation program.

parameters of those components, as well as the map of physical obstacles to generate sensory data for robots.

The basic layer of the environment is represented by the `SimulationEngine` class. This layer has a special meaning, because in addition to generating inputs for collision sensors, it computes positions of robots based on their actions in each cycle. Conceptually, this physical layer is a two-dimensional plane on which robots move. In this text, this plane is referred to as field. Each robot has a body of a circular shape that is placed somewhere on the field. Robots can move through the field by choosing their actions. The `SimulationEngine` is responsible for executing those actions and changing the coordinates of the robots on the field appropriately. The physical world constrains the movement of robots in two ways. First, it defines a global speed limit for robots, a maximum distance a robot can move in a single cycle. Second, it detects and resolves collisions between physical bodies. There are two types of physical bodies in the simulation: robots, and physical obstacles given by a bitmap. If a robot collides with another physical body, its movement stops at that point and its collision sensor receives information about the occurred collision.

The shape of the field is given by a bitmap of obstacles that user provides in a scenario file. As bitmap is an image of a size of `M*N` pixels, the field is a rectangle of a size of `M*N` units. These pixel-units are used as a universal measure of distance in the simulation, not only for field sizes, but also for robot sizes, speeds, etc. Program reads the bitmap and constructs the field. The bitmap is interpreted in such a way, that all the black pixels (`0,0,0` in RGB) are recognized as obstacles and all the other pixels are ignored. So, if a bitmap contains a black pixel at a coordinate `(i,j)`, the field will contain a rectangular physical obstacle at the same coordinate. Because of the way the field is defined, the resulting map of obstacles is discrete even though the field itself is continuous. There is also another way to define the field, without a bitmap. User can just specify a width and a height of the field in pixels in a scenario file; this will create an empty field of a specified size.

Internally, the map of obstacles is represented as an array of line segments. If a robot's action results in crossing any of those line segments, the `SimulationEngine` rolls that robot back in time to the moment when that collision has happened. The same thing happens when two robots collide, in this case `SimulationEngine` computes the point of their collision with respect to the velocities of these robots. Because of this collision resolution mechanism, two physical bodies can never intersect with each other on the field.

The positions of robots in each cycle are computed in floating-point arithmetic. A robot's action consists of two elements: rotation and forward movement. Each element is represented with a double value: a rotation angle in radians, and a speed of movement. A speed of movement can take the values in range from `0.0` to `1.0` (in units of distance per cycle); this limit is enforced automatically even if robot chooses the value outside of it. Based on those values, and taking into account occurred collisions, the `SimulationEngine` computes the positions of robots for the next cycle. After those computations are performed, it is ready to generate inputs for collision sensors.

Collision sensors receive input that consists of a list of collision points, and an action that was taken in this cycle. Collision point is a double value that represents an angle at which the collision has happened, relative to the robot's frontal point. Action forms a part of a collision sensor input, because it can provide a robot with a feedback that tells which action was taken in the last cycle, and how well it was executed. If, for example, the robot was moving forward and then collision has happened, the field `degreeOfRealization` of an action object will be less then `1.0`. This field shows a fraction of a movement that was actually executed.

The `Environment` singleton provides a direct access to generated sensory inputs for robots that have the corresponding sensors. The implementation of this feature is described in the section 2.3.

In addition, this object has several other responsibilities. It is responsible for logging simulation status to console, for writing simulation logs file, and for stopping the simulation. There are two ways the simulation can stop. The regular way is when the end condition is met. In

the scenario file user has to specify the number of cycles that have to be simulated; after that number, simulation stops. It can also stop earlier if `Environment` receives an end signal from `Coordinator` (see section 2.3).

Sometimes an exception can occur during the execution of the JDEECo Runtime. When it happens inside of an ensemble or a component's process, this exception is swallowed by the runtime. To avoid this all the system DEECo entities (`Coordinator`, `RobotEnsemble`, and `ObjectEnsemble`, see section 2.3) use the method `Environment.exitWithException`. This method writes the occurred exception to console, and then exits the program without finishing the simulation. As all the methods of the `Environment` are called from those entities, this prevents swallowing of the exception that can occur in the simulation management code.

2.2 Structure and behavior of DEECo components in simulation

There are two different types of DEECo components that can be present in the simulation. Components of the first type are called robots, they have to extend the `DEECoRobot` class. Components of the second type are called objects, and their common ancestor is the `DEECoObject` class.

Objects are DEECo components that do not interact with physical bodies; they exist in the JDEECo Runtime and can communicate with other components through ensembles, yet they do not have any sensors or actuators to interact with the environment. Each object has several parameters, specified in a scenario file: its coordinates on the field, a size, and a tag string. Because it has coordinates and size, it can be said to exist in the physical environment, however those parameters exist mainly for the visualization, and do not affect the environment or the simulation process in any way (at least by default, user can use those parameters in their own layers of environment, as is shown in the scenario creation guide). An object has an access to all its parameters and can change them directly: to set its coordinates to any value (even place itself outside of the field) or to change its size and tag at any moment. All those parameters are collected by the environment in each cycle of the simulation and are written to the simulation logs.

Robots, on the other hand, have physical bodies, and have to comply with the physical laws of the simulation, namely the speed limit and collisions. Initial parameters of a robot are its coordinates on the field, a rotation angle, a size, and a tag string. Yet, the robot has no direct access to any of those parameters except for the tag string. All the other parameters are stored in the environment and can change only through physical interactions between the robot and the environment. The size of the robot is given at the initialization of the simulation and cannot change. The rotation angle and robot's coordinates can be changed by robot indirectly, by using its wheels to send actions to the environment. In the same way as objects, robots can change their tags at any moment, and in the same way those tags are collected by the environment in each simulation cycle.

Each robot has a field `wheels`. The type of this field is `Wheels`, which is an interface with two methods: `sendCurrentAction` and `setAction`. The first method returns an object of type `Action` and is called by the environment in each cycle. Normally, this method should return an action the robot wants to perform at the moment, yet the definition of this method is up to user that will write an implementation of the Wheels interface. Action object contains two fields that define the desired action: a speed of movement and a rotation angle. Based on those parameters, environment computes position of the robot in the next cycle. Particular implementations of `Wheels` can choose to limit the maximum speed or rotation angle allowed, or to disallow rotation and forward movement at the same time.

User-defined ensembles deployed in the JDEECo Runtime can exchange information between both robots and objects. There are some limits on how a components' attributes can be changed. First, neither ensembles, nor components themselves should change `rID` and `oID` fields that are present on each robot and object respectively. Second, the fields of `Coordinator` object should not be accessed (there are some exceptions described at the end of the section 2.3). These

restrictions are necessary to preserve the integrity of the simulation. Since in JDEECo there is no way to hide those fields from other components, users should be aware of these restrictions.

For robots, knowledge exchange with other components is not the only way they can receive information. They can get data directly from the environment through their sensors. The set of sensors is encapsulated in the field `sensor` of type `SensorySystem`. This class has three methods: `registerSensor`, `unregisterSensor`, and `getInputFromSensor`. Each sensor that is present on the robot is identified by its unique name and type of input that it receives. By default, there is only one sensor on each robot with the name `"collisions"` and type `CollisionData` – the collision sensor associated with the basic layer of environment. User can register additional sensors, each of them has to correspond to an environment layer defined as a separate `SensoryInputProcessor`. As is given by scenario file format, for each environment layer there is a unique name, and a `SensoryInputProcessor` class that returns inputs of some particular type. The name of sensor on robot has to be the same as the name of environment layer this sensor receives data from. Robot can get sensory data from its `SensorySystem` through the `getInputFromSensor` method. This method gets two parameters: a name of the sensor the robot wants to get data from, and an expected type of those data. A `SensorySystem` accesses sensory data through the method `getInputFromSensor` defined on the `Environment`, and returns them to the robot. It works only if the sensor with that name is registered on the robot, the corresponding environment layer exists in the simulation, and the data this layer generates have the expected type.

2.3 The mechanisms controlling the simulation process

The environment and components in the JDEECo Runtime are connected to each other in several ways described below.

The transmission of sensory data from the environment to components is carried out by robots' `SensorySystem`s. At any moment, a robot can request an input from any of its sensors through its `SensorySystem`. It forwards the request to `Environment` where the actual inputs are stored. If there is an environment layer with a specified name, and a corresponding sensor is registered on the robot, it returns a current input.

The remaining connections are provided by the `Coordinator` component and two system ensembles that connect it to other components in the simulation: `RobotEnsemble` and `ObjectEnsemble`. The `Coordinator` has a single process method `cycle`, scheduled periodically with a period of 1 millisecond (which is the minimum possible period). In this method, `Coordinator` regulates the process of the simulation by going through three consecutive phases.

In the first phase the `Coordinator` just waits and counts milliseconds. While it waits, the other tasks, processes of the components and knowledge exchanges, can take place. As different scenarios need different amount of time for all the tasks to occur, the length of this phase can be modified by user. By default it is set to 1 millisecond, the minimum possible time. There can be scenarios in which a complex procedure of decision-making and coordination between the components has to be carried out in each cycle. This procedure may require execution of many processes and knowledge exchanges several times in a row, which cannot happen in a single millisecond of simulated time. In this case, extension of this phase may be useful. Increasing the length of this phase can be viewed as increasing processing capabilities of the components: they can execute more processes in a single simulation cycle.

When the waiting is finished, the `Coordinator` switches to the fetching phase. In this phase, the `ObjectEnsemble` collects the current parameters of the objects (their positions, sizes and tags), and the `RobotEnsemble` gets current tags from the robots. The `RobotEnsemble` also calls the method `sendCurrentAction` on robot's `wheels` to receive an action that robot wants to perform in this cycle. Those wheels are then stored back so that the side effects of the `sendCurrentAction` method could be saved. Both ensembles store the obtained information on `Coordinator`. The fetching phase lasts 1 millisecond, just enough for both knowledge exchanges to occur.

After this, the `Coordinator` proceeds to the last phase, the processing phase. The purpose of this phase is to process the collected component data, and move on to the next simulation cycle. The `Coordinator` stores all the collected data on the `Environment`, and then calls the method `Environment.computeNextCycleAndWriteLogs`. In this method, the `Environment` writes the component data it received to the simulation logs file, computes the positions of robots for the next cycle, and generates inputs for all the sensors by going through the list of environment layers and calling `SensoryInputsProcessor.sendInputs` on each one. After this, the cycle number is incremented, and the `Coordinator` switches back to the waiting phase. From this moment, all the robots have new positions and new data on their sensors, and can make new decisions based on these data. From the point of view of simulation process, the processing phase does not take any time at all, because all the computations happen inside of the `Coordinator`'s process. Therefore, for the components, the shift to the next cycle happens instantly.

The `Coordinator` has two special fields that can be used in the scenario: `status` and `endSignal`. User can access them through knowledge exchange in any user-defined ensemble. The first field is a string that is intended to represent a global status of the simulation. This status is written to simulation logs in each cycle, and during the visualization it is displayed on top of the visualization screen. There are no restrictions on how user can use this field, any value can be written there. The `endSignal` field is a boolean value that can be used to end the simulation. The simulation will stop the next cycle after a user-defined ensemble will set this value to `true`.

2.4 Initialization of the simulation

The central entry point of the program is the `Simulation` class. When user runs the program, an instance of this class is created and initialized with the provided scenario file. It validates the file against the corresponding XML Schema, and then uses it to load, create and initialize all the entities described there. The format of the scenario file is described in detail in user manual that can be found in the project's repository.

Simulation instantiates all robots, objects, and additional layers of environment that are specified in the scenario file. For each created entity it calls two initialization methods: `setParameters` and `processArg`. The first method initializes attributes of an entity to the values that are specified in the scenario file. It has default implementation that can be overridden if user wants to process those values differently. The second method receives a string parameter specified by a user in the scenario file, and does nothing by default. Users can override this method to interpret the string in some way. An example of usage of this feature is demonstrated in the chapter **??**.

When all the entities are initialized, the `Simulation` checks the consistency of the initial parameters: all robots should be placed inside of the field, and their initial positions should not contain collisions. If the initial parameters are inconsistent, or if the scenario file contains mistakes, a `SimulationParametersException` is thrown.

All those steps are executed in the constructor of the `Simulation` class. After a `Simulation` object is instantiated, the program calls `Simulation.startSimulation` method. This method initializes the two main subsystems, and starts the simulation process. At the end of this process user has a new simulation logs file ready for visualization.

When the method returns, these steps are repeated for the next scenario, if more scenario files were provided.

3 Simulation logs format

Simulation logs are written in a plain text format. Their format is described by the grammar below.

```
Legend:
    NONTERMINAL_SYMBOLS
    terminal_charachters
    terminal_symbols (variables)
```

```
SIMULATION_LOGS =
    BITMAP_SECTION
    ---
    ROBOTS_SECTION
    ---
    OBJECTS_SECTION
    ---
    BODY

BITMAP_SECTION =
    null
    width_of_the_field
    height_of_the_field

BITMAP_SECTION =
    width_of_the_field
    height_of_the_field
    bitmap_encoding

ROBOTS_SECTION =

ROBOTS_SECTION =
    fully_qualified_name_of_robot_class,robot_size
    ROBOTS_SECTION

OBJECTS_SECTION =

OBJECTS_SECTION =
    fully_qualified_name_of_object_class
    OBJECTS_SECTION

BODY = CYCLE

BODY =
    CYCLE
    BODY

CYCLE = status&&LIST_OF_ROBOTS&&LIST_OF_OBJECTS

LIST_OF_ROBOTS =

LIST_OF_ROBOTS = ROBOT;;LIST_OF_ROBOTS

ROBOT = x_coordinate,,y_coordinate,,angle,,tag

LIST_OF_OBJECTS =

LIST_OF_OBJECTS = OBJECT;;LIST_OF_OBJECTS

OBJECT = x_coordinate,,y_coordinate,,object_size,,tag
```

The first three sections constitute a header, which is written at the initialization of the environment. It is followed by a body that is appended each cycle with a single line that contains the current values of components' parameters. Parameters `x_coordinate`, `y_coordinate`, `angle`, `robot_size`, and `object_size` are double values; parameters `status` and `tag` are strings; `width_of_the_field` and `height_of_the_field` are integers.

Ampersand, semicolon, and comma are metacharacters that separate individual sections of the cycle line in this format. Because of this, if any of those symbols is encountered in status string or in a tag, the symbol gets prefixed with a backslash. When Visualization program reads simulation logs, those backslashes are removed.

If a bitmap was provided in the scenario file, then the bitmap section will contain a bitmap encoding. A bitmap is encoded as a sequence of zeros and ones: `1` for a black pixel, `0` for a non-black one. Based on this code, visualization program reconstructs the original map of the field.

All the values written in a simulation logs file are used in some way by the Visualization program.

4 Visualization

The Visualization program interprets simulation logs files to produce an animation that demonstrates the behavior of the cyber-physical system that was simulated. When a user starts the Visualization program, a window appears on the screen, in which the scenario is drawn cycle by cycle. Users can interact with the visualization using keyboard. The following commands are supported by the program:

- pause/resume (space bar),
- rewind/fast forward (left/right arrow keys),
- increase/decrease the speed of visualization (up/down arrow keys),
- restart visualization from the beginning (enter key).

The systems that can be simulated by the framework consist of four different types of entities that could visualized: robots, objects, maps of physical obstacles, and additional layers of the environment. Different scenarios will need to display these entities differently, and some of them will have additional information that has to be displayed. To address the needs of the various scenarios, Visualization program uses the concept of customizable visualization layers.

The rendering routine iterates through a list of objects that extend `VisualizationLayer` abstract class; these objects are responsible for drawing individual shapes and textures on the screen. By default, visualization contains three layers: map layer, robot layer, and object layer. The first one draws a map of physical obstacles, rendering them as black squares. The second one draws robots as red circles, and the third one is responsible for drawing objects; they appear as yellow squares by default. For visualization of many complex scenarios, these settings will not be sufficient. For this reason, the framework allows users to create custom visualization configurations.

Configurations are written in XML files of a specific structure. The format of these files is described in the user manual that can be found in the project's repository.

In the attributes of the root element of a configuration file, several global parameters of the visualization can be set. Those parameters are: zoom, speed of the visualization (in cycles per second), speed of the rewind/fast forward, and a color of the status string.

Apart from the global parameters, a configuration file contains a list of visualization layers that have to be visualized, with instructions on how to visualize them. The order in which layers appear in a configuration file is important. Layers are drawn one by one in that order. If,

for example, a node corresponding to the object layer is written after the one that specifies the robot layer, objects will be drawn on top of robots. By default, the first layer is the obstacle layer, then follows the object layer, and the last one is the robot layer.

If some of these default layers is not specified in the list, it is appended at the end of it implicitly. Thus, a configuration file can have an empty list of layers. In this case, the three default layers will be added implicitly, in the default order, with the default settings. The only impact of a configuration file like this will be on the global visualization parameters. However, if a user needs to hide a particular layer, there is an ability to make a layer transparent.

Robots, objects, and physical obstacles can have colorings specified by a user. A coloring is either a texture or a color in which an entity will be drawn. A color can be specified in two ways. The first one is as a comma-separated list of 3 or 4 float values. These values are interpreted in RGB or RGBA models respectively. The second one is with an identifier (e.g. `"red"`, the full list of supported identifiers can be found in the manual). A texture has to be specified as a path to an image file.

The settings of the obstacle layer are simple, they can contain only a single coloring. The settings of the robot and object layers (collectively referred to as component layers) are more complex, since they allow to draw the components individually.

Each component can have its own coloring, or even change its coloring during the visualization. There are three parameters based on which the coloring can be chosen: an individual number, a class, and a tag. User can choose one of these options for each component layer. With a number option, a coloring is chosen individually for each component depending on its number in the list of robots or objects. A class option allows to choose a coloring for all components of the same class. With the last option the program will draw components differently based on tags they currently have. Each of the options allows also to change the default coloring. The default coloring is applied to the components which do not have a specific coloring assigned.

Several other options can be set in the settings of the component layers. User can turn on displaying individual numbers or tags of the components. The color of the displayed text also can be chosen. For robots, rotation can be enabled; in this case their textures will reflect their current rotation angles. Also in these settings user can choose to draw objects as circles instead of squares. In this case, textures for objects will be rounded (for robots this is done by default).

The default layers are not the only layers that user can include in the list. There are two other types of layers: background layers and additional layers. A configuration can contain any number of these layers.

A background layer displays a single coloring (thus, a color or a texture) on the whole visualization window. Mostly, there is no reason to have more than one layer of this type, except for the partially-transparent textures.

An additional layer is a layer that can be fully defined by a user, by extending the `VisualizationLayer` abstract class. This class has a method `render` that has to be implemented. It receives a number of a cycle that is currently visualized, and has to draw all the shapes and textures that correspond to this layer. It also has a hook method `processArg` which is called at the initialization of the layer. This method receives a string argument, specified in the configuration file.

There are two additional methods defined in this class: `drawText` and `loadTexture`, which can be helpful for the implementation of an additional layer. It also contains references to objects that can draw shapes and render textures on the screen. An example of the implementation of additional layer is shown in the scenario creation guide.

Figures 3 and 4 show an example of how the visualization can be altered with a configuration file.

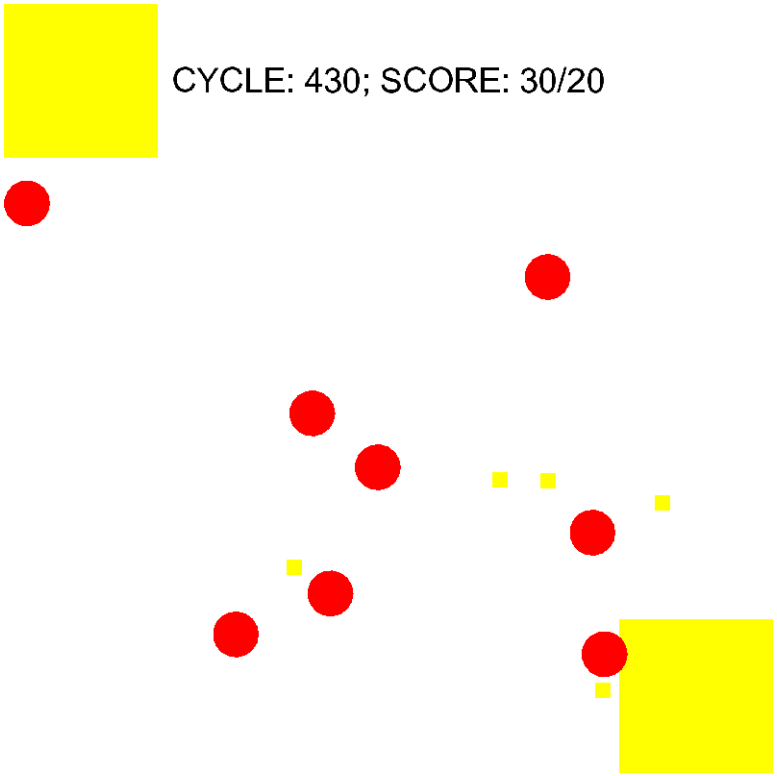

Figure 3: An example scenario visualized with default configuration

Figure 4: An example scenario visualized using a custom configuration file.