A virtual playground for DEECo applications

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Petr Hnětynka, Ph.D.
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2017
I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .......... date ............ signature of the author
DEECo is a novel component model for designing software-intensive cyber-physical systems. As a practical realization of this model there was developed the JDEECo implementation, written in Java. The usage of this framework was shown in a number of different scenarios. However, at the moment there are only limited ways to systematically create, simulate, and visualize new scenarios written with this model. This thesis presents a virtual playground that allows to create scenarios featuring autonomic robots programmed in DEECo. The playground offers a number of options in creation of scenarios, including programming robots and their interactions, customizing and extending the physical environment, and adding interactive objects. These scenarios can also be visualized with the developed application. The parameters of visualization can be customized for needs of a specific scenario. The functionality of the application is demonstrated on several example scenarios.

Keywords: Cyber-physical systems, simulation, visualization, DEECo
I would like to thank my supervisor, doc. RNDr. Petr Hnětynka, Ph.D., for his guidance and valuable advice. I would also like to thank my girlfriend Anastasia for her support throughout my work on this thesis.
Contents

1 Introduction 3
   1.1 Cyber-Physical Systems .................................. 3
   1.2 DEECo component model .................................. 4
   1.3 Goals of the thesis ...................................... 4
   1.4 Thesis organization ...................................... 5

2 Problem Analysis 6
   2.1 Implementation of the DEECo component model in the JDEECo framework ......................... 6
      2.1.1 Representation of the first-class concepts .............................................. 6
      2.1.2 How the JDEECo runtime framework works .............................................. 8
   2.2 Connecting DEECo components to the environment .............................................. 10
   2.3 Representation of the physical environment .................................................... 11
   2.4 Options in scenario creation ................................................................. 13

3 Application Structure 15
   3.1 Simulation .................................................... 16
      3.1.1 Environment subsystem ......................................................... 16
      3.1.2 Structure and behavior of DEECo components in simulation ......................... 19
      3.1.3 The mechanisms controlling the simulation process .................................. 21
      3.1.4 Initialization of the simulation ...................................................... 22
   3.2 Simulation logs format ............................................................................. 23
   3.3 Visualization ......................................................................................... 25

4 Scenario Creation 29
   4.1 Creating a simple scenario ........................................................................ 29
      4.1.1 Adding an additional environment layer ............................................... 29
      4.1.2 Programming wheels ........................................................................... 31
      4.1.3 Programming robots and ensembles ................................................... 31
   4.2 Creating a complex scenario ...................................................................... 31
      4.2.1 Layers of the environment .................................................................... 32
      4.2.2 Firefighters’ wheels ........................................................................... 33
      4.2.3 Behavior of firefighter robots ............................................................. 34
1. Introduction

1.1 Cyber-Physical Systems

The term Cyber-Physical Systems (CPS) refers to a broad range of diverse and complex systems that integrate digital computing, network communications, and physical processes [7]. In contrast to traditional hardware/software systems, CPS operate in inherently dynamic physical environment. They use sensors to collect data from the environment and actuators to act upon it. More complex CPS consist of many autonomous and interdependent components communicating and cooperating with each other to carry out a specific task. In recent years, CPS have found applications in a number of different areas. Examples of CPS include embedded systems in vehicles, industrial control systems, medical monitoring devices, etc. As domestic robots start appearing in households and millions of sensors populate the streets of modern cities, importance of cyber-physical systems further increases.

As CPS increase in size and complexity they become more and more software dependent, meaning that software becomes their most sophisticated and important constituent. However, due to the differences between CPS and traditional software systems, common principles of software engineering are hard to apply in the design of CPS [4]. One frequent problem is testing and validating software for CPS. Since physical processes and interaction with them play an important role in CPS, it is hard to test the software because of the costs associated with running a whole system in physical environment. To make the process of development less expensive and more dynamic it proves to be useful to create a virtual environment where virtual representations of components could be simulated along with the physical world. A virtual environment of this kind can enable developers to test and refine software for CPS before it becomes ready for deployment on actual physical components.

Virtual environments for simulations of CPS can also serve another important function. If the results of the simulation can be visualized, then such a tool can be used to observe the designed system at early stages of the design process and to study the emergent behavior of the system. Those visualizations can also demonstrate the functionality of the system to those who are unfamiliar with its internal organization. To perform these functions, a virtual environment could model only the basic attributes of the physical environment, leaving aside insignificant details. This thesis presents a virtual playground that enables to simulate a particular kind of cyber-physical systems and to visualize their behavior. This virtual playground uses the DEECo component model as a basis for systems that it can simulate.
1.2 DEECo component model

DEECo (Distributed Emergent Ensembles of Components) is a novel component model designed specifically to target the development of software-intensive CPS [1]. It is built around two core first-class concepts: components and ensembles. Components are autonomous entities living and operating in dynamic environment. Each component is defined in terms of information it possesses, called knowledge, and processes that determine its behavior. Processes can be understood as periodically scheduled or triggered procedures operating on component’s knowledge that may have sensing or actuating as side effects. Ensembles are implicit groups in which components can be organized. Each ensemble is formed dynamically between the components that satisfy its membership condition to execute a procedure called knowledge exchange [2]. In the same way as processes, knowledge exchanges operate on the knowledge of components, but they can communicate knowledge of one component to another. Ensembles constitute the only way of direct communication between components in the DEECo model.

As a practical realization of the DEECo model, there was created the JDEECo framework which provides a Java implementation of the DEECo concepts [5]. This framework allows to create DEECo-based applications by implementing the components and ensembles as Java classes. These classes then can be deployed into the JDEECo runtime environment that manages the scheduling and execution of their procedures.

A practical use of this framework was presented in a number of test case scenarios [6]. Also, there exist a few demonstrators featuring simple examples of the usage of this framework. Those demonstrators have mostly textual output. The are exceptions, like the Convoy tutorial, which presents a simple animation accompanying the execution of the JDEECo runtime. However, at the moment there is no simple way to create, simulate and visualize new scenarios written using the DEECo model. The DEECo model could benefit from a tool that would enable to do this.

1.3 Goals of the thesis

The goal of this thesis is to design and develop an application that will serve as a virtual playground for autonomic robots programmed in DEECo. This application will enable users to write custom scenarios for simulation, run simulations of those scenarios, and visualize the results of those simulations. The main purpose of the application will be to provide a tool for creating demonstration scenarios for the DEECo component model. The application has to offer users a wide range of options, that would enable to create different kinds of scenarios.

Simulation and visualization will be designed as separate parts of the appli-
cation. The results of simulations will be stored in a format suitable for the visualization. The application will receive scenario descriptions in XML files of a specific structure.

1.4 Thesis organization

The following chapter 2 provides a technical background to the JDEECo implementation, and an analysis of the problems the application faces.

Chapter 3 describes the implementation of the virtual playground that was developed.

A guide to creation of scenarios for the developed virtual playground is presented in the chapter 4.

The digital attachment accompanying this thesis contains the application itself, including all the libraries needed to build and run it, several example scenarios demonstrating functionality of the program, and a user manual. The manual describes all the options that users have in writing scenarios for the application, and how to use these options.
2. Problem Analysis

The chapter presents an analysis of the requirements for the virtual playground. It discusses challenges and considerations that had an impact on the design decisions made in the process of the development of the application.

The first section of the chapter provides a technical background to the JDEECo runtime framework. First, it concentrates on the user perspective: how DEECo component and ensemble classes look like, and how to create them. Second, it discusses some details about the framework’s implementation. As JDEECo is a complex framework, only those details are discussed that are relevant for the implementation of the virtual playground and for creation of scenarios for it. The application uses version 3.0 of the JDEECo framework, which is the most recent stable version of the framework at the moment [6].

The subsequent sections discuss the problems the application needs to solve: what types of connections have to be established between the components in the JDEECo runtime and the simulation of the external environment, and what are the requirements for the environment itself. Some of the main features of the developed solution are described, leaving the implementation details for the chapter 3.

2.1 Implementation of the DEECo component model in the JDEECo framework

2.1.1 Representation of the first-class concepts

In JDEECo, both components and ensembles are represented as Java classes of a specific structure. JDEECo uses a domain-specific language implemented via Java annotations to mark those entities and their elements. A component is a class with the @Component annotation. All the public non-static fields of this class are interpreted as a component’s knowledge nodes. These nodes can be read and written by the component’s processes. Each public static method marked with the @Process annotation is interpreted as the component’s process. Processes can be understood as functions that operate on a component’s knowledge, and cannot have any side effects on a component’s state or on the states of other components.

A process can have multiple inputs and outputs, that have to be specified in the parameters of a corresponding method. Each parameter of a process method has to be annotated with one of the following annotations: @In, @Out, or @InOut. Based on these annotations, the runtime framework decides whether to retrieve a knowledge node before the execution of a process, and whether to store it back afterwards. The execution of a process can be either periodically scheduled, or
triggered by a change in the component’s knowledge. In the first case, the method is annotated with the `@PeriodicScheduling(period)` annotation, where `period` is an integer that shows a number of milliseconds that have to elapse between the two consecutive executions of the process. A triggered scheduling can be employed by attaching the `@TriggerOnChange` annotation to one of the input parameters of a process. Then, this process is executed whenever the value of that knowledge node is changed.

An example of the component definition is shown below.

```java
@Component
public class DEECoRobot {

    public String id;
    public SensorySystem sensor;
    public Wheels wheels;

    @Process
    @PeriodicScheduling(period = 1)
    public static void makeDecision(
        @InOut("wheels") ParamHolder<Wheels> wheels,
        @In("sensor") SensorySystem sensorySystem
    ) {
        // decision-making code
    }
}
```

A string argument that each of the `@In`, `@Out`, and `@InOut` annotations takes refers to a name of the node in the component’s knowledge that is provided to this process. For `@Out` and `@InOut` annotations that knowledge node has to be wrapped in a `ParamHolder` object.

Ensemble classes are marked with the `@Ensemble` annotation. Each ensemble performs a knowledge exchange, which can be understood as a process that can take knowledge nodes of two components as parameters at the same time. In order to participate in a knowledge exchange, a pair of components has to satisfy an ensemble’s membership condition. Therefore, two public static methods have to be present in an ensemble class. The first method contains a membership condition. It is marked with the `@Membership` annotation, and returns a boolean value. The second one is a knowledge exchange method, which is annotated with `@KnowledgeExchange`. Each parameter of those methods has to be marked with one of the same `@In`, `@Out`, or `@InOut` annotations. However, the parameters of membership condition method can be marked only with `@In`. An example of ensemble is shown in the listing below.
Each of the two components that participate in a knowledge exchange is identified by a role it plays in an exchange. One component takes a role of the ensemble’s coordinator, and another one takes a role of its member. These roles do not have any predefined semantics, they only define which knowledge nodes are taken from which component in this particular ensemble. As the listing shows, the names of the knowledge nodes that an ensemble takes as parameters have to be prefixed with either "coord" or "member" identifier, which points to a component a knowledge node is taken from. To participate in a knowledge exchange, two components need to have all the knowledge nodes required by an ensemble. Thus, even though a membership condition in the example above returns "true" for each pair, to play a coordinator role in this ensemble a component needs to have a field phase of type Phase, and a to play a member role it needs a field rID of type Integer. Thus, these requirements can also work as a filter for some potential pairs of components.

2.1.2 How the JDEECo runtime framework works

Components are deployed to the JDEECo runtime as instantiated objects. When a component gets deployed, the runtime framework extracts all its knowledge fields and stores them in the knowledge repository. An extraction of a knowledge node is basically a deep cloning of its internal structure. All the objects referenced from an extracted node get cloned. Therefore, several components inside the runtime cannot have references to a same object, because this object is cloned separately when each of the components gets deployed. Since components are
autonomous entities, all their knowledge exists locally.

During the runtime execution, processes and ensembles operate on the knowledge stored in the knowledge repository. Processes and ensembles are collectively referred to as tasks. At the runtime initialization, each periodically scheduled task is added to a list of tasks for execution. After it is executed, it is re-scheduled for the next execution based on its period.

JDEECo uses single-threaded scheduling and execution of tasks. The scheduler has a queue of tasks, each scheduled at some particular moment. The time that this scheduler works with is a simulated time, which has no relation to the actual execution time. From the point of view of the simulation timer, the tasks happen instantly, no matter how much real time an execution of a task has taken.

When a component gets deployed, a ProcessTask object is created for each process method defined in the component’s class. At a basic level of understanding, an execution of a task of this type happens in three simple steps. First, all the input knowledge nodes are retrieved from the knowledge repository. Then, the method corresponding to the task is invoked with these knowledge nodes as parameters. After the method returns, all the output values are stored back to the knowledge repository.

Ensembles are handled by the runtime framework differently. Ensembles are deployed as classes instead of instantiated objects. When an ensemble class is deployed, a separate EnsembleTask is created for each component in the runtime environment. All those tasks are scheduled and executed independently. Each of them has a component it is associated with. When an ensemble task is executed, it puts the component into both a role of coordinator and a role of member consecutively. For each role, it iterates through a list of all the other components, and invokes a membership condition for the pair, putting each component in the list into the opposite role. If a membership method returns true for a pair, a knowledge exchange method is invoked for it. After this method returns, only those output values are stored back to the repository that belong to the component associated with this particular EnsembleTask.

This makes sense, again, because components are autonomous entities. For each deployed ensemble, a component has a corresponding EnsembleTask. This task scans the available knowledge of other components to form a pair, but is executed locally, on the component it is associated with, so only the outputs that belong to this component are stored back. The output values of the other component will be stored when a task corresponding to the same ensemble will be executed on it. This way of execution, however, can sometimes cause problems.

Let us say that component C1 and component C2 exchange their knowledge through an ensemble E, which has output values for both components. In this case, two EnsembleTasks exist: E-C1 and E-C2. After the first task is executed, the knowledge of C1 is stored back, yet the changes to the knowledge of C2 are not saved. When the E-C2 task takes place, C1 has different values in its knowledge.
(due to the previous exchange), and the execution of this task will have different results. The changes to the knowledge of C1 can even cause that the membership condition of E will no longer hold for this pair, thus for the C2 this knowledge exchange will never take place.

This implementation detail is worth discussing, because it had an impact on the design of the virtual playground, and has to be considered in the design of scenarios for it. It is impossible to know in advance which of the E–C1 and E–C2 tasks will take place first, or which tasks, if any, will be executed between them. Therefore, ensembles have to be designed in a way that will account for this asynchronous nature of knowledge exchange.

Thus, not only all the knowledge that components possess is local, but also all the tasks are. We can imagine that each component stores its knowledge and runs its tasks on its own hardware. Even knowledge exchanges with other components are local, as they result only in a change of knowledge of a single component.

### 2.2 Connecting DEECo components to the environment

The JDEECo runtime environment is a closed system that manages the behavior and interactions of autonomous components. To use this runtime environment as a basis for creation of virtual simulations of CPS, the components need to be provided with an access to a virtual simulation of the physical environment. A simulated physical environment like this has to be global, in a sense that all components have to be a part of a single environment. The components have to be able to communicate with one another not only through ensembles, but also through physical interactions. Each component has to have access only to its partial view of the environment that it receives through its sensors. Components also have to be able to act upon the environment in a way that is limited by capabilities of their actuators.

A virtual simulation of a physical process can be created by breaking the continuous physical process into discrete steps, called cycles. Then, a physical process can be computed by iterating the execution of an algorithm that computes a state of the physical system in the next cycle based on its state in the previous one. To connect a physical system like this to the JDEECo runtime environment properly, it needs to be ensured that the computations of the next cycle are synchronized with the JDEECo timer.

As it was stated in the section 1.3, the visualization of the simulated system has to be independent from the simulation itself. To make this possible, the simulation has to log its state regularly and to store these logs in external files. A visualization program then, can read the files written by a simulation program, and interpret them to visualize the simulated scenario.
Parts of the state of the simulation are distributed across the knowledge of all the simulated components. However, the logging of this state needs to be centralized. Therefore, there have to exist at least two centralized subsystems in the application: the external environment subsystem and the logging subsystem. Since both of these subsystems have to be connected in some way to all the components in the runtime, they can be merged together to avoid the duplication of the connections. Thus, the application needs to establish proper connections between the components distributed in JDEECo runtime and the centralized subsystem responsible for simulating the external physical environment and writing the simulation logs. Later in the text, this subsystem is simply referred to as the environment.

Thus, to ensure the correct flow of the simulation process there have to exist the following connections between the components in the JDEECo runtime on the one hand, and the centralized environment subsystem on the other:

1. Transmission of sensory data from the environment to components. These data are generated by the physical environment in the process of computing the state of the physical system. Each component has to have an access to them at any moment. The knowledge of the components that reflects the state of the environment has to be updated every time when the state of the environment changes.

2. Collection of data from robots’ actuators, and transfer of these data to the environment. The environment has to be informed about the actions of components in each cycle to compute the state of the physical system correctly.

3. Collection of the current information about the state of the components, and transfer of these data to the environment. This refers to the information that is needed for the visualization. This has to occur regularly so that the relevant information could be written into the simulation logs file.

4. Control of the simulation cycle. Computations of the state of the environment and generation of new sensory data for robots have to be executed regularly so that the simulation would advance. Yet, components and ensembles should have enough time for their tasks to take place in each cycle.

The implementation of these connections developed for the application is described in detail in the section 3.1.3.

2.3 Representation of the physical environment

The other challenge in the development of the application is the creation of the physical environment itself. There exist a number of physics engines that can
simulate physical interactions. Most of the freely available physics engines are targeted on the game development, and produce real-time computer graphics. The particular choice greatly depends on the goals of the application, and its desired characteristics.

The virtual playground is meant to be mostly a demonstration tool. It has to offer users ability to create and test demonstration scenarios based on the DEECo component model, rather than to design high-precision virtual prototypes of the actual DEECo-based cyber-physical systems. The selection of the physics solution for this virtual playground was influenced by the following requirements and considerations:

1. Separation of an external and an internal views on the environment. Robots should not have access to the environment itself, only to the data on their sensors. These data should be generated by the environment for each robot individually. The knowledge nodes that correspond to the sensory data have to be updated implicitly by the environment, after the computations of each simulation cycle.

2. Separation of simulation and visualization phases. In the process of simulation, the environment has to compute interactions between the physical bodies, and write the results into the simulation logs. The simulation logs are intended to be visualized later, without running the JDEECo runtime or re-simulating physics. Users have to have control over the visualization process: to pause/resume the visualization, increase/decrease the speed of it, or to rewind it back.

3. Extensibility. As different scenarios need to simulate different aspects of the physical world, the environment has to be easily extensible by users. Extension of the environment has to be simple and straightforward, so that any scenario could make use of this feature.

4. Simplicity. The development of the complex physics engine would take a lot of time, as well as solving the problem of connecting and adapting the existing complex engine to the JDEECo runtime. In addition, if users of the application would need to learn the details of the complex physical engine, it will make the creation of the scenarios difficult and will limit the usage of the application.

5. Compatibility with JDEECo. It should be possible to connect the computations in the environment with the components in JDEECo runtime in a way that was described in the previous section.

These considerations have ruled out the idea of using an existing physics engine. It was decided to create a simple 2D simulation of the physical environment, that will manage the interactions between robots’ bodies. The environment
contains a map of physical obstacles that user can specify in a bitmap. To make things simple, both for development and for users, it was decided to limit the possible shapes of robots’ bodies to circles of different sizes. Introducing additional shapes would make the development of physics engine unnecessary expensive, and would add little in terms of the range of possible scenarios that this application would enable to create. It is also much harder to write programs controlling the behavior of the robots of arbitrary shape, and this would increase the difficulty of writing the scenarios featuring the additional robot shapes.

For the visualization of simulation logs that the developed environment produces, it was decided to use libGDX graphical library [8]. libGDX is a free and open-source multiplatform game development framework implemented in Java. The virtual playground uses the features this framework provides for the work with graphics and for processing input from keyboard.

2.4 Options in scenario creation

The usefulness of the virtual playground will depend greatly on the number and variety of scenarios it will enable users to create. For this reason, the environment described in the previous section has to be easily extensible by users.

The actual physical world is very complex, and can be a source of different kinds of information. There exists a large number of different types of sensors that can provide robots with this information: acoustic, visual, thermal, etc. A simulation of the physical world can therefore be divided into several layers, each working with its own kind of information. This notion of layers is implemented in the application.

By default, there exists only a single basic layer of the environment, that manages the physical interactions between robots. Users can extend the environment by defining additional environment layers, and providing robots with the sensors that correspond to those layers. This application makes the process of creating additional layers very simple. The implementation of this feature is described in the section 3.1.1 and examples of its usage are shown in the chapter 4.

To further increase the range of scenarios that the playground enables to create there was added a concept of non-physical objects. These objects are also DEECo components in the same way as robots, yet they are not constrained by the physical laws of the simulation, and do not have sensors or actuators to interact with the physical world.

Thus, the virtual playground has to offer users the following options in creation of scenarios:

1. Loading new classes for robots (physical DEECo components).
2. Loading new classes for objects (non-physical DEECo components).
3. Loading new classes for ensembles.

4. Drawing maps of physical obstacles.

5. Adding new layers of environment, and new sensors that can receive information from those layers.

6. Defining initial positions and physical sizes of robots in simulation.

7. Programming robots' actuators.

Additionally, the visualization has to be extensible as well. Since the application will allow to create a wide range of scenarios, users have to be able to modify the visualization settings. The application has to allow not only to alter the appearance of existing entities (robots, objects, physical obstacles), but also to add additional visualization layers defined by user.
3. Application Structure

The chapter presents a complete overview of the virtual playground’s structure and implementation.

As it was mentioned earlier, the application is divided into two separate programs: Simulation and Visualization. Those programs are executed independently, and the only thing that connects them to a single application 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 3.1 shows a general scheme of interaction between the user and the application.

![Diagram showing inputs and outputs of the application.]

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 digital attachment to this thesis.

3.1 Simulation

The main idea behind the application was to use the existing JDEECo Runtime and to connect it to a virtual simulation of external environment. The components living in JDEECo Runtime should be able to 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 3.2.

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.

3.1.1 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 Sensory-InputProcessor 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 chapter 4. 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. SIPs can access the lists of robots and objects and the parameters of those components, as well as the map of physical obstacles to generate sensory data for robots.
Figure 3.2: The general scheme of the simulation program.
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 $M \times N$ pixels, the field is a rectangle of a size of $M \times N$ 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 3.1.3.

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 3.1.3).

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 3.1.3) use the method Environment.exitWithException. This method writes the occurred exception to console, and then exits the application 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.

3.1.2 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 chapter [4]). 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 [3.1.3]). 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, unregis-
terSensor, 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.

3.1.3 The mechanisms controlling the simulation process

The environment and components in the JDEECo Runtime are connected to each other in several ways. The list of the required connections is given in the section 2.2.

The transmission of sensory data from the environment to components is carried out by robots' SensorySystems. 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.

### 3.1.4 Initialization of the simulation

The central entry point of the application is the Simulation class. When user runs the application, 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 digital attachment to this thesis.

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 4.

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, application 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.

### 3.2 Simulation logs format

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

Legend:

```
NONTERMINAL_SYMBOLS
terminal_characters
terminal_symbols (variables)
```

```
SIMULATION_LOGS =
  BITMAP_SECTION
  ---
  ROBOTS_SECTION
  ---
  OBJECTS_SECTION
  ---
  BODY
```
The first three sections constitute a header, which is written at the initializa-
tion 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 \texttt{x\_coordinate}, \texttt{y\_coordinate}, \texttt{angle}, \texttt{robot\_size}, and \texttt{object\_size} are double values; parameters \texttt{status} and \texttt{tag} are strings; \texttt{width\_of\_the\_field} and \texttt{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.

### 3.3 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 application 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 \texttt{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, application 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 digital attachment to the thesis.

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 section 4.2.4.

Figures 3.3 and 3.4 show an example of how the visualization can be altered with a configuration file.
Figure 3.3: An example scenario visualized with default configuration

Figure 3.4: An example scenario visualized using a custom configuration file.
4. Scenario Creation

The chapter serves as a guide to creation of scenarios for the application. 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 digital attachment to the thesis.

4.1 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.

4.1.1 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.

```java
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:

```xml
<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.

```java
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`):

```java
CollisionData collisionData = sensor.getInputFromSensor("collisions", CollisionData.class);
Coordinates coordinates = sensor.getInputFromSensor("coordinates", Coordinates.class);
```
4.1.2 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.

```java
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:

```java
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.

4.1.3 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.

4.2 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 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.

4.2.1 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.

### 4.2.2 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.

4.2.3 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 Data-AggregationAndOrderDistribution 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.

4.2.4 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 4.1 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 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 4.2 and 4.3 show the resulting picture.
4.3 Using other features of the application

Besides the two described scenarios there are several others included in the digital attachment. 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 chapter 3.

The Competition scenario, pictured on the figure 3.4 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 application 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.
4.4 Related work

There has been already done some work on visualization of DEECo-based systems. The JDEECoVisualizer application \cite{9} targets visualization of smart transportation infrastructure. It allows to visualize outputs of simulations made with Multi-Agent Transport Simulation Toolkit (MATSim). This toolkit supports development of scenarios in which entities move through a map of interconnected nodes (like vehicles moving through a city). The JDEECoVisualizer application displays the components and the ensembles present in the scenario that was simulated with MATSim.

The virtual playground presented in this thesis aims at another goal. Its main purpose is easy creation of scenarios. It offers a number of options for this purpose, and allows users to compose many different types of scenarios with these options. All the scenarios simulated with this playground can be visualized, and the visualization settings can be modified for each specific scenario.
5. Conclusion

The goal of the thesis was to design and develop a virtual playground for autonomous robots programmed in DEECo. The playground is meant to be a tool for creation of demonstration scenarios for the DEECo component model.

It was decided to create a simple and easily extensible simulation of physical environment, in which the robots would operate. Robots receive information from the environment through their sensors, and can act upon it using their actuators. The set of their sensors can be extended, and their actuators can be implemented in different ways. To expand the variety of scenarios that can be simulated with the playground, there were added non-physical objects, as an additional type of DEECo components.

The scenarios for the playground can be composed simply by specifying all the used classes and their initial parameters in an XML file of a simple structure. The results of simulations are stored in simulation logs files, which can be visualized. Users can control the process of visualization with keyboard. The visualization settings can be customized to reflect the specifics of the visualized scenario.

Thus, it can be concluded that the application fulfills the goal stated in the section 1.3 and offers all the options listed in the section 2.4. The functionality of the application was demonstrated in several example scenarios described in the chapter 4.

5.1 Future work

There are several potential directions for future work. The existing physics engine could be improved to support more complex physical interactions, additional shapes of robots, etc. Introducing these features could require to alter the format of simulation logs and to make some changes in the default visualization layers, however the remaining structure of the application could be preserved. The general architecture of the Simulation program could be preserved even for creation of more sophisticated 3D simulations. Of course, the programs controlling the behavior of robots in that case would be much more complex, and the development of scenarios would be much more difficult. However, that would open another direction for usage of that application: high-precision prototyping of actual CPS.

The other feature that could be added to the application is real-time visualization of the simulated system. Real-time visualization has many drawbacks comparing to the chosen solution. The complexity of a simulated scenario can significantly impact the speed of simulation, and for this reason there would be great difference in the speed of visualization of different scenarios. In addition, it would not be possible to control the process of visualization in that case. However, there could be reasons for using this feature. Also, the real-time visualization could
complement the existing process: it is possible to visualize the simulated system
and to write the simulation logs at the same time.

Even though this application was designed with an intention to offer users as
much options as possible, it is likely that there could be many additional features
that could be added to the application to improve its usability. However, even the
existing set of features allows to create a wide range of scenarios for the DEECo
component model.
Bibliography


# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Inputs and outputs of the application.</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>The general scheme of the simulation program.</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>An example scenario visualized with default configuration</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>An example scenario visualized using a custom configuration file.</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Firefighters scenario with the default visualization configuration.</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Firefighters scenario visualized with a configuration file written specifically for it</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Firefighters scenario: robots #1 and #2 are broken; robot #3 is in danger and returns to the charging station.</td>
<td>37</td>
</tr>
</tbody>
</table>