AATOM - An Agent-based Airport Terminal Operations Model Simulator

Stef Janssen
s.a.m.janssen@tudelft.nl
Delft University of Technology

September 2017

1 Introduction

This document gives an overview of the functionalities of the AATOM simulator. AATOM is Java-based and is used to simulate airport terminal operations in an agent-based way. It provides the user with a large set of basic functions useful for experimentation. This document is outlined as follows. An installation guide is found in Section 2, a set of tutorials is provided in Section 3. Finally, frequently asked questions are stated in Section 4. The full Java documentation for the simulator can be found at https://stefjanssen.github.io/AATOM/. Furhtermore, the code used in this work can also be found on Github.

2 Installation

AATOM can be downloaded from https://github.com/StefJanssen/AATOM. After downloading, include the .jar file into the path of a java project in the IDE of your choosing. A tutorial for Eclipse can be found here, for Intellij it can be found here.

The .jar file contains all functionality of AATOM and can be extended to create your own models in. It also contains some basic example airports that you can simulate yourself.

Basic introductions to Java are readily available from the web. You can for instance take a look here or here. Refer to the tutorials section below for ways to use AATOM.

3 Tutorials

This section contains a set of tutorials that show the basics of AATOM. The tutorials gradually increases in complexity, until you designed your own airport. Code for all tutorials are provided on Github.

3.1 The first simulation

AATOM contains two airports that can be used for simulation: Eindhoven Airport and Rotterdam The Hague Airport. To simulate Eindhoven Airport, use the following code in the main method of your project. This main method you have to create yourself. If you do not know what this means or how to do this, please refer to the Java introductions websites of the previous section.

```
tutorial1();
```

The first command (on line 1) in this method generates a Simulator called *sim* from the ModelBuilder class. This class contains the prebuilt simulations for Eindhoven Airport and Rotterdam The Hague Airport. The inputs are a Boolean value stating if a GUI should be displayed and an integer stating the time step of the simulation (in milliseconds). The command on line 2 creates a thread and starts the simulation.

To be able to run this code, you need to import the correct classes in your own class. In an IDE such as Eclipse or Intellij, you can automatically do this by resolving the errors it shows. If multiple classes can be imported, a general rule of thumb is to choose the class that does *not* start with *'java.'*.

When the program is run, you see a visualization of Eindhoven Airport. In addition, passengers are generated and execute different activities in the airport. Processes like check-in and security are also present. Replacing *eindhovenAirport* by *rotterdamTheHagueAirport* in the code on line 1 generates a simulation for Rotterdam The Hague Airport.

3.2 Building the Environment

You can design your own environment by adding *environment objects* to the simulation. An example is given on how to add a wall to the simulation, but all environment objects are added in the same way. An overview of existing environment objects can be found in the documentation.

```
tutorial2a();
```

The first line creates a simulator, much like in the tutorial above. In this example, the base ending conditions are used and the GUI is set to true. These ending conditions state that the simulation should end after a specified number of seconds. In the example this is 20 seconds.

Next, on line 2, a wall is added to the simulator. The input arguments for the wall are the top left x coordinate, the top left y coordinate, the width and the height in meters respectively. Finally, on line 3, a threat is created and the simulation is started the same way as in the example above. Running the simulation shows a single wall for 10 seconds, after which the simulation is stopped.

We extend this example by creating a queuing area.

```
tutorial2b();
```

Line 3 shows how a queue is created and added to the simulator. A queue is essentially a collection of queue separators and a queue area that are generated by the *ModelComponentBuilder*. This is why *addAll* is used: more objects are added to the simulator at the same time. The first parameter of the queue method specifies the top left corner position of the queue, the second parameter specifies the number of horizontal lanes. The third parameter specifies the width of the queue, and the fourth parameter adds a 'blocking wall' to the simulation. Finally, the last parameter specifies the rotation of the queue. Running this simulation shows a wall (the one we saw before) and a queue leading towards that wall.

3.3 Creating the First Airport

This tutorial shows how to create your first airport in AATOM. As in the previous examples, first, we create a simulator with the base ending conditions and a GUI. Further, we create a map builder, as was done in the previous examples as well.

Then, we create a gate area at the top left corner of our map, followed by a set of chairs in the gate area. Then, a check-in area and checkpoint area is generated. These are accessed from the map builder, and include all necessary parts for simulation of these elements. The next lines are used for the creation of a flight in the environment. The flight needs a number of input arguments that are found on 11-16, while the creation and addition of the flight to the simulation happens on lines 17-19. Line 20-21 creates a passenger and adds it to the simulation. Finally, line 22 starts the simulation. This is all included in the following method.

```
tutorial3();
```

When you start the simulation, you see a single passenger that moves from check-in, to the checkpoint and finally sits down on one of the chairs in the sitting area. The passenger is implemented following the AATOM architecture. You can create your own implementation of a passenger by extending the passenger class and extending the different layers (strategic, tactical and operational). Each of these layers has its own modules that need to be defined.

3.4 Working with the Agent Generator

This tutorial extends the tutorial described above ('Creating the First Airport'). To work with the agent generator, replace line 1 of this tutorial by the lines below. Further, remove lines 20-21 from the example. This is the call that starts with *sim.add(new Passenger(...))*. This is already done in the next method.

```
tutorial4();
```

The agent generator generates an agent with expected inter-arrival times of 30 seconds. The generator ensures that a random flight is chosen from the set of flights that leave in at least 30 minutes and at most 3 hours. You can create your own agent generator by extending the existing AgentGenerator class.

3.5 Adding Analyzers

This tutorial extends the tutorial described above ('Working with the Agent Generator'). Here, we add analyzers to the simulation. Analyzers show data of the simulation over time. Add the lines below right before you start the simulation.

```
sim.add(new QueueAnalyzer());
sim.add(new TimeInQueueAnalyzer());
sim.add(new ActivityDistributionAnalyzer());
sim.add(new TimeToGateAnalyzer());
sim.add(new AgentNumberAnalyzer());
sim.add(new MissedFlightsAnalyzer());
sim.add(new DistanceAnalyzer());
```

This is already done in the following method.

```
tutorial5a();
```

By adding these analyzers, you see graphs in the Graphs tab of the visualization. On top of that, the simulator automatically logs the graphs you add to a .txt file. See Section 3.7 for more information on how to analyze this data.

You can also create your own analyzer to analyze some parameters of the simulation. You can do so, by extending the abstract class Analyzer. In this example, we create an analyzer that tracks how many check-in operators are currently active. To extend the Analyzer, you have to create a new file called *'TutorialAnalyzer.java'* and fill it with the content below.

```
public class TutorialAnalyzer extends Analyzer {

@Override
public String[] getLineNames() {
     return new String[] { "# of operators active" };
}
```

```
@Override
public String getTitle() {
     return "# of check-in operators active";
}

@Override
public double[] getValues() {
     double numberOfActiveOperators = 0;
     for (OperatorAgent operator :
     getSimulator().getMap().getMapComponents(OperatorAgent.class)) {
         if (operator.getAssignment() instanceof OperatorCheckInActivity) {
             if (!operator.getActiveActivities().isEmpty())
                 numberOfActiveOperators++;
         }
     }
     return new double[] { numberOfActiveOperators };
}

@Override
public String getYAxis() {
    return "# of operators active";
}
}
```

Four methods need to be implemented to extend the Analyzer class. Three methods are used to give names to various elements of the analyzer (the title, the lines and the y axis), while the fourth is used to determine the values of the lines. In this example, a single line is used to indicate the number of active check-in operators. Make sure you add it to the simulation as well by adding the line below in your main code.

```
sim.add(new TutorialAnalyzer());
```

This is already done in the following method.

```
tutorial5b();
```

3.6 Creating Your Own Passenger

In this tutorial we will extend the standard passenger class to create our own passenger. This class can be created in a different .java file called *TutorialPassenger.java*. The only difference with the current passenger is that this extended passenger also logs a '1' if it is sitting. To create this passenger, use the code showed below.

```
public class TutorialPassenger extends Passenger {

    public TutorialPassenger(Map map, Flight flight, boolean checkedIn,
        Class<? extends Facility> facility,Position position,
        double radius, double mass, Luggage luggage, Color color) {
        super(map, flight, checkedIn, facility, position, radius, mass, luggage, color);
}
```

```
    public void update(int timeStep) {
         super.update(timeStep);
         if (isSitting())
              setLog(new String[] { "1" });
    }
}
```

This code consists of two parts: a constructor and an update method. The constructor is needed to create the passenger, while the update method specifies the behaviour of the passenger. The constructor has only one function: passing parameters to the super (Passenger) class. The update method consists of two parts. The first part (*super.update*(*timeStep*)) makes sure the standard passenger behaviour is executed, while the second part logs a 1 if the agent is sitting. As we only created the Tutorial Passenger, and did not yet add it to the simulator, the behaviour will not yet show. To achieve this, we extend the current BaseAgentGenerator class so that TutorialPassengers are added to the simulation. The code (created in a different *TutorialAgentGenerator.java* file) to do this, is shown below.

```
public class TutorialAgentGenerator extends BaseAgentGenerator {

public TutorialAgentGenerator(double interArrivalTime) {
    super(interArrivalTime);
}

@Override
public HumanAgent generateAgent(long numberOfSteps, int timeStep, boolean forced) {
     if (forced || canGenerate(timeStep)) {
              Luggage luggage = new Luggage(LuggageType.CARRY_ON,
              Utilities.RANDOM_GENERATOR.nextDouble(),
              Utilities.RANDOM_GENERATOR.nextDouble());
         if (areas.isEmpty())
             return null;
         EntranceArea area = areas.get(Utilities.RANDOM_GENERATOR.
         nextInt(areas.size()));
         Position start = Utilities.generatePosition(area.getShape());
         Flight flight = getEligibleFlight();
         if (flight != null) {
             return new TutorialPassenger(simulator.getMap(),
             flight, false, null, start, 0.2, 80, luggage,
             Color.RED);
         }
    }
return null;
}
}
```

The constructor passes an argument to the super class, while the *generateAgent*(*numberOfSteps*, *timeStep*, *forced*) method ensures that a TutorialPassenger is generated with random parameters if possible.

We now adapt the previous tutorial to ensure that the tutorial agents generator is used, instead of the base agent generator.

```
Simulator sim = new Simulator(true, 100, new BaseEndingConditions(3600),
new TutorialAgentGenerator(30));
```

This is already done in the following method.

```
tutorial6();
```

After running this example, you will see that the *agentLog.txt* file is not empty anymore and contains the logged data.

3.7 Analyzing Data

By default, data of the different analyzers is saved in a text file. To add the traces of each agent to the log, a different type of logger has to be used. This can be done by changing the previous example as follows.

```
tutorial7();
```

After a simulation run, a collection of log files are generated in a subfolder of the *'logfiles'* folder. This folder can be found in the main folder of your Java project by default. For each simulation run, a folder that is named based on time is generated. Four files can be found in this folder: *'agentLog.txt', 'returnValues.txt', 'agentTrace.txt'* and *'trackedParameters.txt'*. These files contain the logs that you added yourself, the values returned by the EndingConditions, position history of passengers and the data of the analyzers respectively. These text files can be imported with your favorite data analysis tool. Matlab code to read and save the .txt file is provided in the analytics section on Github. These Matlab scripts and functions can be used to import and analyze data generated by AATOM.

To use this code, run the *'importAndVisualize.m'* script in Matlab and select the *folder* that contains the log files of a simulation run. So for instance, select *'logfiles/1503561764574_9956'*, and not *'logfiles/1503561764574_9956/agentTrace.txt'*. This script visualizes the data of the *'agentTrace.txt'* and *'trackedParameters.txt'* files. Further, it saves all data into a data format that can be used for further analysis.

3.8 Customized Views

You can add customized views for MapComponents that you created. In this example, we create a custom view for the TutorialPassenger as defined in Section 3.6. To do this, we extend the MapComponentView class in a new TutorialPassengerView class, as shown below.

```
public class TutorialPassengerView extends MapComponentView {

private TutorialPassenger passenger;

public TutorialPassengerView(TutorialPassenger passenger) {
    this.passenger = passenger;
}

@Override
public String getAboutString() {
    return "<html><i>Hello</i> world, my hashcode is: " + passenger.hashCode() + "</html>";
}

@Override
public void paintComponent() {
    ShapeDrawer.drawCircle(Color.GREEN, passenger.getPosition(), passenger.getRadius());
    // Set the bounds for the about box.
    setBounds(ShapeDrawer.getRectangle(new Position(passenger.getPosition().x
```

```
    - passenger.getRadius(), passenger.getPosition().y - passenger.getRadius()),
    2 * passenger.getRadius(), 2 * passenger.getRadius()));
}
}
```

To create a custom view for a MapComponent, the class name should exactly match the class name of that MapComponent, with an addition 'View' at the end. Furthermore, the constructor takes exactly one argument, which is the MapComponent itself. Then, the *paintComponent*() method has to be implemented. This method can be implemented using the *ShapeDrawer* class, which automatically draws shapes at the right place of the GUI.

Finally, mapComponents can be clicked in the GUI. When they are clicked, an about box is shown if the *getAboutString*() method was implemented. For this method to function correctly, the *setBounds*(*r*) method has to be called in the *paintComponent*() method. This method sets the bounds of the mapComponent, such that it generates an about box when clicked within this box. Finally, the *getAboutString*() has to be filled in with any String that you would like to show in the about box. You can format the String following HTML standards to format the about box. In our example, we show show a green circle as a representation for the TutorialPassenger, and show the hash code of the agent in the about box.

3.9 Running Experiments

This final tutorial shows how you can perform experiments with the simulator. To be able to do this, you have to create a new main method and class. This main method starts a simulation like we have done in all previous tutorials. The main difference with the tutorials before, is that we use the *args* field of the main method. This args field can be used to pass arguments to the simulator to specify the experiment conditions. In our case, we extend the tutorials above by adding two input arguments. The first argument specifies the interarrival time, while the second argument specifies the random seed that is being used. For an experiment, we of course do not need visualization, so we turn the GUI off. Finally, we set the name of the simulation to the two input arguments, separated with an underscore. This name will be the last line of the 'agentLog.txt' file, and will later be useful to analyze the simulation outcomes. A snippet of the class is shown below.

```
public class ExperimenterMain {

public static void main(String[] args) {
    int interarrivalTime = Integer.parseInt(args[0]);
    long seed = Long.parseLong(args[1]);
    Simulator sim = new Simulator.Builder<>().setSimulationName(args[0] + "_" + args[1])
         .setAgentGenerator(new TutorialAgentGenerator(interarrivalTime))
        .setEndingConditions(new BaseEndingConditions(3700)).setGui(false)
        .setLogger(new BaseLogger(true)).setRandomSeed(seed).build();
    [...]
    }
```

This class is now used to start the experiment using the Experimenter class. This class takes three input arguments: a list of parameters to experiment with, the main class and the number of parallel tasks to be ran. The third argument defaults to the number of available cores in the machine.

```
List<String[]> inputs = new ArrayList<>();
inputs.add(new String[] { "30", "111111" });
inputs.add(new String[] { "30", "222222" });
inputs.add(new String[] { "40", "111111" });
inputs.add(new String[] { "40", "222222" });
```

```
Experimenter experimenter = new Experimenter(inputs, ExperimenterMain.class);
new Thread(experimenter).start();
```

4 Frequently Asked Questions

Frequently asked questions are shown here. If you have a question that this document does not answer, please contact the author of this document. Contact details are provided at the top of this document.

How can I zoom in and out in the GUI?

Zooming in works by dragging your mouse over a specific area. When you do so, you will see a black rectangle appear that shows to area that indicates the area that your will zoom into. By clicking on the right mouse button, you zoom out to the original view.

Why can't I see the logged files of my simulation?

There are a few common causes and corresponding solutions for this problem.

1. You forcefully closed the simulation. This means you pressed *ctrl + c* in a command window or closed the program from the task manager of your operating system/IDE. To prevent this from happening, ensure that your ending conditions are defined properly, close the simulation using the GUI, or wait until the simulation run has ended.
2. You explicitly mentioned that the simulator should not log your simulation by using a *null* parameter in the constructor of the simulator. To prevent this, remove the *null* parameter from the constructor.
3. You do not have writing permission in the operating system. To solve this, run the program in administrator mode.
4. The simulation is still writing the last logs. Depending on the computing speed of your computer, it can take a short while after you have closed the simulation before the .txt files are ready to be opened.

Can I edit the Eindhoven Airport example of the first tutorial?

You cannot edit the example that is present directly, however you can make a copy of the source code for this example and edit this copied code. The code for this example can be found on Github. Find the eindhovenAirport method, and copy it into a class of your choosing. You can now change the example.

How can I generate multiple agent types in my own agent generator?

You can do so by making sure there are multiple return statements (returning different agent types) in your *generateAgent*() method. Based on your own defined conditions, one return statement returning the first agent type can be reached, while other agents can be reached based on other conditions.