UbikSim 2.0 Developers Manual

University of Murcia

Juan A. Botía - *juanbot [at] um.es*
Pablo Campillo - *pablocampillo [at] um.es*
Francisco Campuzano - *fjcampuzano [at] um.es*
Emilio Serrano - *emilioserra [at] dit.upm.es*

November 18, 2014

Listings

Contents

1 Basic controls

This section describes how to create a simple example of simulation with UbikSimIDE.

1.1 Installing and executing UbikSimIDE

Firstly, it is required to install Java 3D. For this purpose, it is recommended to follow the instructions of the Java 3D website[1].

There are three different ways to execute UbikSimIDE:

- A project using UbikSimIDE.jar, its javadoc and the presentation "Developing social simulations with UbikSim" can be found in this link [2]. This is the most recommended distribution to learn the basic concepts of UbikSimIDE. It is a Java proyect for Netbeans. However, it also could be opened with any other Java IDE. If you are using Netbeans: *File, Open Project* and select the folder which contain the downloaded distribution of UbikSimIDE.
- If you want to have the full source code of UbikSimIDE in the project mentioned in the point above, remove the file ./lib/UbikSimIDE.jar and add the src of the IDE available here [3].
- If you want to work with UbikSimIDE and include your changes in this repository:
  - Install the gitHub plugin.
  - Clone the UbikSim repository link [4].
  - Download and include in your local folder third party libraries, environments, configuration files and NetBeans files from this [5].
  - Now you can use UbikSim as a NetBeans project (compile it, run it, etc.). Besides, the repository contains a .gitingore file to keep you from committing files which are not the source code.
  - Don't forget to sync after making commit in your local repository

By default, the main class of the UbikSimIDE project is *"UbikEditor"*. Then, execute the UbikSimIDE project and the UbikSim editor will appears, see figure 1(a). Within the editor, it is possible to load a previously created simulation file (.ubiksim) or to create a new simulation model. If the Ubik Editor is not shown or you have problems with the Java3D libraries, please revise the Java 3D installation and check if the UbikSimIDE project has access to these libraries.

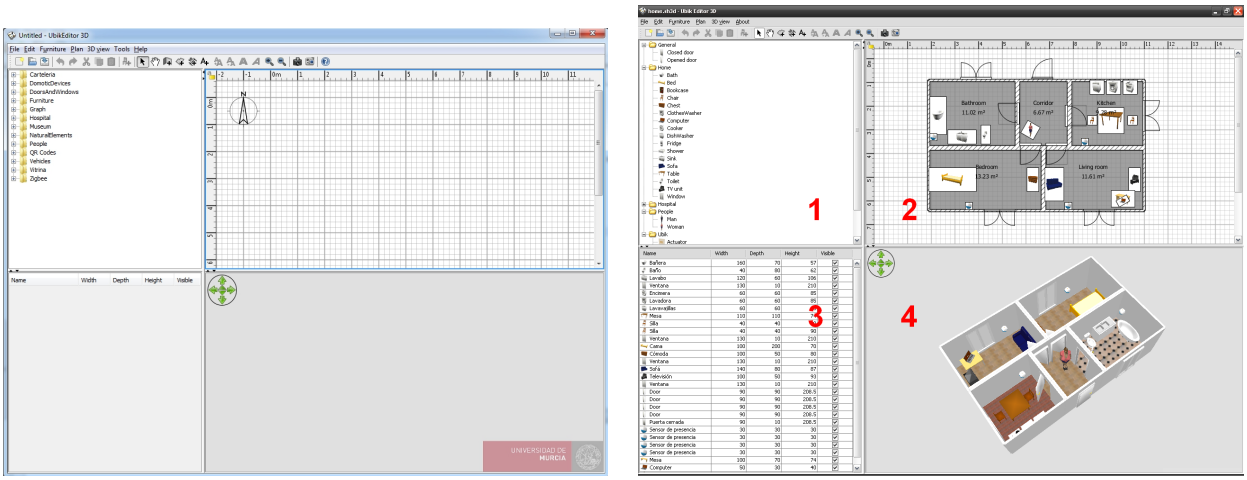
1.2 How to use Ubik Editor to create a simulation

A tutorial to create a basic simulation follows. This tutorial is concerned to learn only the basic controls. If you are interested in create more complex models, it is recommended to study the section 2 of this document.

Ubik Editor is an indoor environments editor integrated in UbikSimIDE. This editor allows the users to create buildings over a 2D plan, and also offers a navigable 3D view of the model. There exists the possibility of including a wide range of furniture, domotic devices and persons. Figure 1(b) shows an screenshot of the editor which is divided in four resizable panes.

- The part 1 is the objects palette. It contains all the available furniture, domotic devices, persons, doors, etc. you may add to your model, they are classified in folders. Every object of this palette must have an associated JAVA class if it is going to manifest behaviours during the simulation.
- The part 2 is the 2D view. Over this plan viewed from top, the user can draw the rooms and walls of the environment with the mouse and layout the objects. The objects to be included in the model can be dragged & dropped to the plan directly from the palette. The position of the objects can be modified at every moment. A magnetism function allows to put the objects fixed to other objects, e.g. a clock fixed

[1] `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html`
[2] http://ants.dif.um.es/staff/emilioserra/ubiksim/UbikSimClass
[3] https://github.com/emilioserra/UbikSim/archive/master.zip
[4] https://github.com/emilioserra/UbikSim/
[5] http://ants.dif.um.es/staff/emilioserra/ubiksim/libAndEnv/UbikSimIDE.zip



(a) (b)

Figure 1: (a) UbikSim editor recently launched, (b) UbikSim showing a simulation model and divided in parts

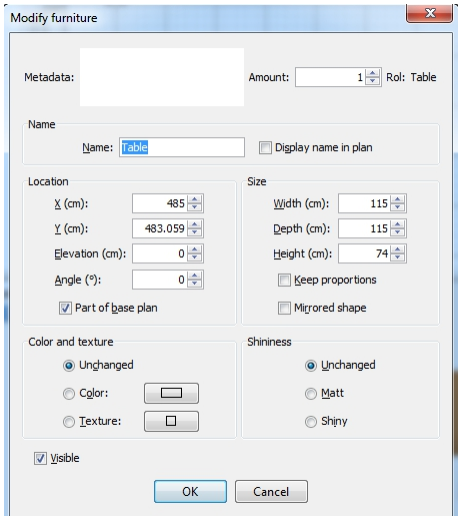


Figure 2: Object properties window

to a wall. Doing double click on an object, it is possible to define some configuration properties as the size or the elevation from the floor. Figure 2 shows the object properties that can be modified. The field metadata allow the users to create specific properties for every object.

- The part 3 is a list which contains all the objects introduced in the model. Their name, size and other characteristics may be displayed. It may be sorted by clicking on each column title. Double-clicking on them it is possible to modify all their properties in a menu as the one shown in figure 2.

- The part 4 is a 3D view of the model. It shows a preview of the model and it is updated in real time when the user modifies the model. Every time a new object is added to the 2D plan or the environment is modified, the preview is updated. The 3D preview has visualization capacities as zoom, rotation and first person view.

From now on, we offer an step by step example of how to create a simple environment. Firstly, for creating the walls click on the button boxed in red in figure 3. A window showing tips about the wall creation will appear (figure 4(a)).

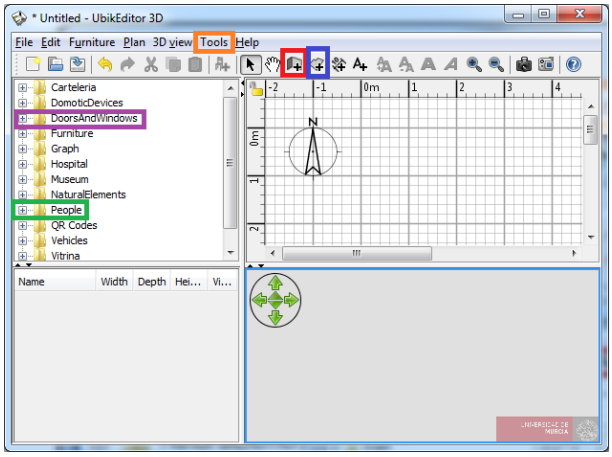
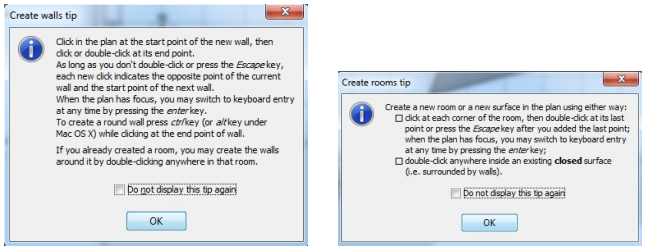
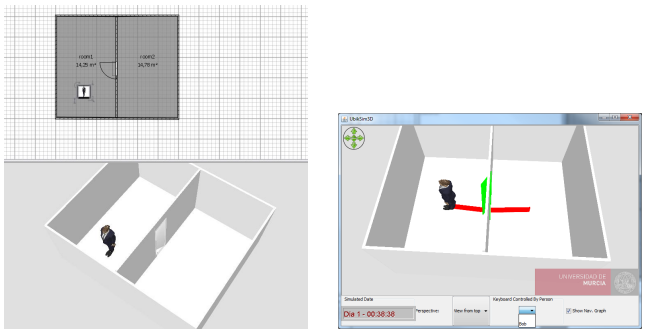


Figure 3: UbikSim editor details



(a) (b)

Figure 4: (a) Wall creation tips, (b) Room creation tips



(a) (b)

Figure 5: (a) Result for a simple environment creation, (b) UbikSim 3D Display

Create two squares with the walls. Then, you can create the rooms clicking on the button boxed in blue in figure 3. A window showing tips about the room creation will appear (figure 4(b)). You can draw the contour of two rooms in this model, one room per every wall square previously created. After that, put a door in the wall positioned between the two rooms. You can select the object *"opened door"* situated into the folder *DoorsAndWindows* of the palette (purple box in figure 3).

The next step is to include a person in the model. It is recommended to use the object *"Teacher 2"* from the folder People (green box in figure 3) and put it in any of the rooms.

Finally, doing double click on the rooms and the person, you can assign them names. These names are useful for reference them in the JAVA code. For example, assigns room1 and room2 to the rooms and Bob to the person. The final result must be similar to this one shown in figure 5(a).

Save the environment as *"experiment.ubiksim"* (File, save as).

1.3 How to launch a simulation within the editor

This section explains how to launch simple simulations within the editor. To know more about how to execute UbikSim with more advanced features, it is recommended to take a look to the section 3 of this document.

Once your simulation file is saved, go to the menu *Tools* (orange box in figure 3) and choose the option *Simulate*. Then, the UbikSim console will appear, see figure 6(a). It is based on the original console of MASON.

The MASON console offers different options. For example, the tab Model (see figure 6(b)) allows to configure some simulation parameters, as the random seed or the cell size or the initial date of the simulation. In the tab Console it is possible to configure some console parameters, e.g., a delay that establishes the simulation speed (recommended between 0.5 and 1), see figure 6(c). It is possible to show the visualization of the simulation in a new window, in 2D or 3D, selecting the option in the tab Displays (figure 6(d)). Finally, you can start the simulation clicking the play button.

UbikSim2D display can be used watch the simulation but its main functionality is to inspect properties of entities in the simulation such as agents and devices. By double-left click on a entity, its properties are presented in the Inspectors tab, see Figure 6. More details about how to use inspector tab at `http://cs.gmu.edu/~eclab/projects/mason/docs/tutorial0/index.html`.

UbikSim3D display offers some functionalities, see Figure 5(b):

- Change perspective of the camera between "View from top" and "View from observer". Use WSAD keys to control de camera in "View from observer" mode.
- Any character can be controlled using keyboard and mouse. The display offer a list with the names of the people, once one is selected, click the 3D frame. Keys to control the character are presented in 1.
- Last option ("Show Nav. Graph"), displays in the scenario the navigation graph, i.e., the paths that agents follow. It is useful to debug scenarios and detect isolated areas.

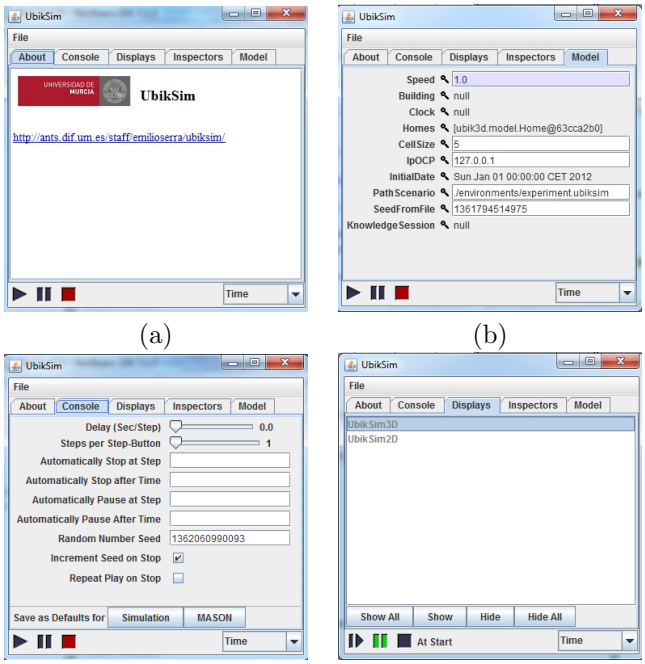


Figure 6: (a) UbikSim console based in MASON, (b) Model configuration tab, (c) Console configuration tab, (d) Display configuration tab

'j' - turn left
'l' - turn right
'i' - go ahead
'k' - 180° turn
'o' - open or close a nearby door
'u' - start or stop person activity
(useful for detecting activity with presence sensor
although the person is not moving)
'+' - increase speed by 2
'-' - decrease speed by 2
'0' - set state of the person to 0
'1' - set state of the person to 1
'2' - set state of the person to 2
'3' - set state of the person to 3
'4' - set state of the person to 4
'5' - set state of the person to 5
'6' - set state of the person to 6
'7' - set state of the person to 7
'8' - set state of the person to 8
'9' - set state of the person to 9

Table 1: Keys to control a person.

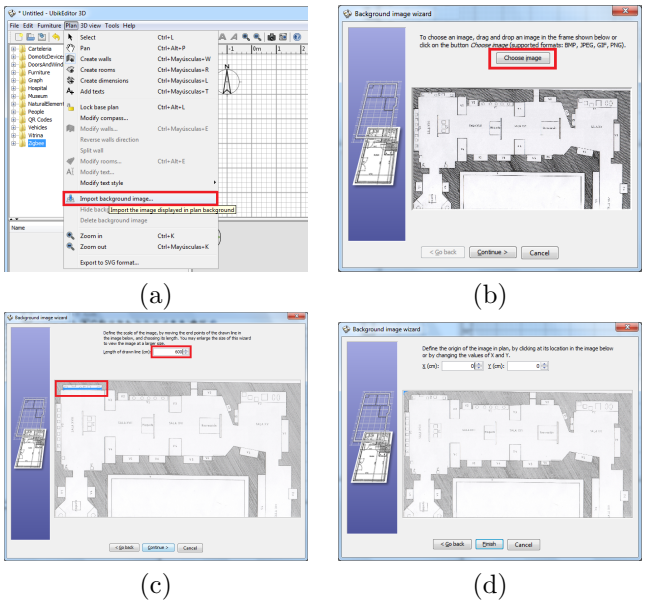


Figure 7: (a) How to import a background image, (b) Wizard to import background image: how to choose an image, (c) How to establish the image scale, (d) How to put the image as background

2 How to create environments with UbikSimIDE

This section explains how to create an environment model with UbikSimIDE and some tips to ease that purpose.

2.1 Creating the environment

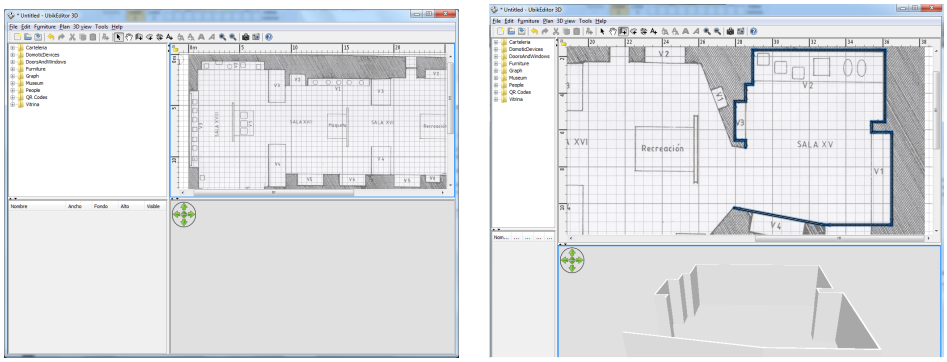
In order to facilitate the environment creation, UbikSimIDE allows the user to import the blueprint of the environment to create. It can be established as the background of the 2D plan window choosing the option *Plan* → *Import Background Image...*, as it is shown in figure 7 (a).

A wizard that helps you to choose and scale an image file will appear. Then, click on *Choose Image* and choose the image file that contains your blueprint in the file dialog. BMP, JPEG, GIF or PNG formats are supported. Once the image is loaded, click on *Continue>*, see figure 7 (b).

The next step is defining the image scale. It is recommendable to enlarge the window to have more precision. Define the scale of the image by moving the end points of the coloured line drawn in the image, in such a way that this line matches a known length. We recommend to choose a distance as large as possible. Once the line is correctly positioned on the image, type the real length of this line in the *Length of the drawn line* field, and click on *Continue*. See figure 7 (c) to see an example.

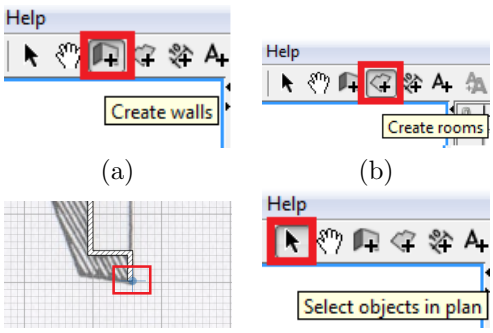
The last step consists in establishing the origin of the image. Define the origin of the image in the plan, i.e. the point in the image matching the point (0, 0) in the home plan. See figure 7 (d). Click on the button *Finish*. Once the wizard is closed, your image will appear behind the home plan grid at the chosen scale, as shown in figure 8 (a). If you chose a wrong scale or location, edit them by choosing *Plan* → *Modify background image...* out of the menu.

The next step is the wall creation. Click on the button *Create walls*, as in figure 9 (a). Then click on the home plan at the start point of the new wall, then click or double-click on the plan at its end point. As long as you don't double-click or press the Escape key, each new click indicates the opposite point of the current wall and the start point of the next wall. It is recommendable to use the zoom tool, with combination Ctrl+<mouse scroll>, in order to achieve more precision. The walls are straight lines, so you can click at every room corner to create them. The wall layout has some predefined angles, probably not enough sufficient in complex environments like this one. To create a wall with any angle hold pushed the *Shift* key. It is also recommendable to do wall layouts as large as possible, e.g. if you are modelling a house, draw first the external walls and later the internals. When the walls are created, it is possible to see in the 3D view the wall creation in real time, see figure 8 (b).



(a) (b)

Figure 8: (a) Background image loaded, (b) 3D view with the walls successfully created



(a) (b) (c) (d)

Figure 9: (a) Tool for wall creation, (b) Tool for room creation, (c) Blue circle that eases the room creation, (d) Tool for object selection

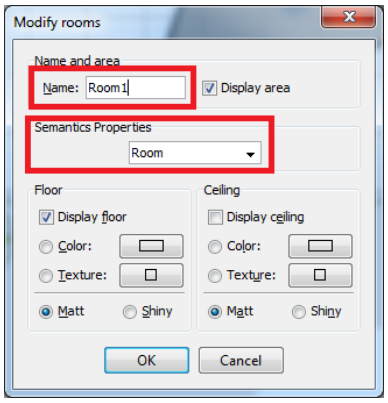


Figure 10: Room properties menu.

Once the walls are created, the next step is the room creation. For that purpose, click on the button of figure 9 (b). To create the rooms click at each corner of the walls and draw the room shape. Then double-click at its last point or press the Escape key after you added the last point. Note that if you situate the cursor near to the wall corners, a blue circle appears, see figure 9 (c). After the creation of the room, employ the objects selection tool, as it is shown in figure 9 (d). Doing double click on the room it is possible to edit its properties (figure 10). In this case, edit the room name, *Room1*, and the room type, *Room*. The room type can be chosen from a combo box that shows all the room types which are implemented in the simulator. It is possible to edit the room type but you must assure that this type is implemented and its name is added to the space handler, *handlers/space_areas.txt*. Note that every room must be completely delimited by walls. If you are interested in not showing a wall, it is recommended to put a frame door whose size was the same as the wall. More information about this type of doors in the next section.

Once the walls and the rooms are created, the next step is the door creation.

2.2 Doors creation

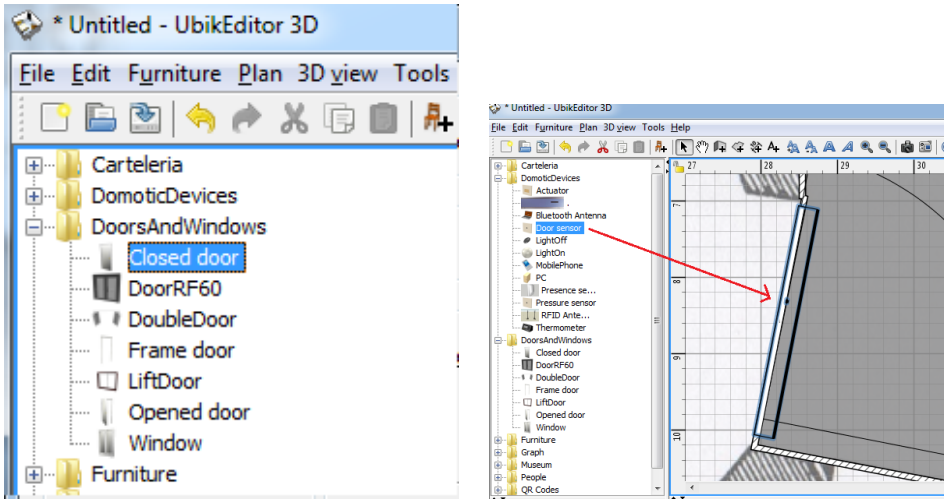
Doors are objects situated at the object palette, in the folder *DoorsAndWindows*, see figure 11 (a). To add furniture (e.g. doors) to your home, drag and drop the corresponding object from catalog to the home plan. Situate them on the walls, you will see how the door recognize the wall and adapts to it. As an example, we will use two type of doors. A *Frame door* is a door without door, only the frame. It is useful to use this object to connect rooms when there is not a door between them in the reality. The second door is the most common, *Opened door*. Once it is situated on a wall, it is necessary to adjust the door width, especially if you are using a frame door and you want to occupy all the wall. It is easy to do this clicking on the object (every object in UbikSimIDE has some icons at each corner of the selected piece, you can use these icons to modify some properties of the object as its size, elevation and angle). See figure 12 for an example of how to make the door wider.

Note that the doors always must connect two rooms. The doors that connect the building with the exterior, are special cases. You must create a room that represents the street for a correct performance of the simulator.

Optionally, the properties of the door can be modified double-clicking on the object, see figure 13 (a). By default, all the doors in the simulator (not in the editor) which are opened are coloured in green. If you want to create closed doors or any other state, you can indicate this at the field *Metadata*. The possible door states are the following, use exactly these names in the field *Metadata* to achieve this kind of door.

- opened: opened door (default state), these doors are coloured in green during the simulation.
- closed: closed without key, these doors are coloured in yellow during the simulation.
- locked: closed with key, these doors are coloured in red during the simulation.

Figure 13 (a) shows an example of how to indicate the door is initially closed, by writing the word “closed”.



(a) (b)

Figure 11: (a) Palette folder containing doors and windows, (b) How to situate a door sensor in a door

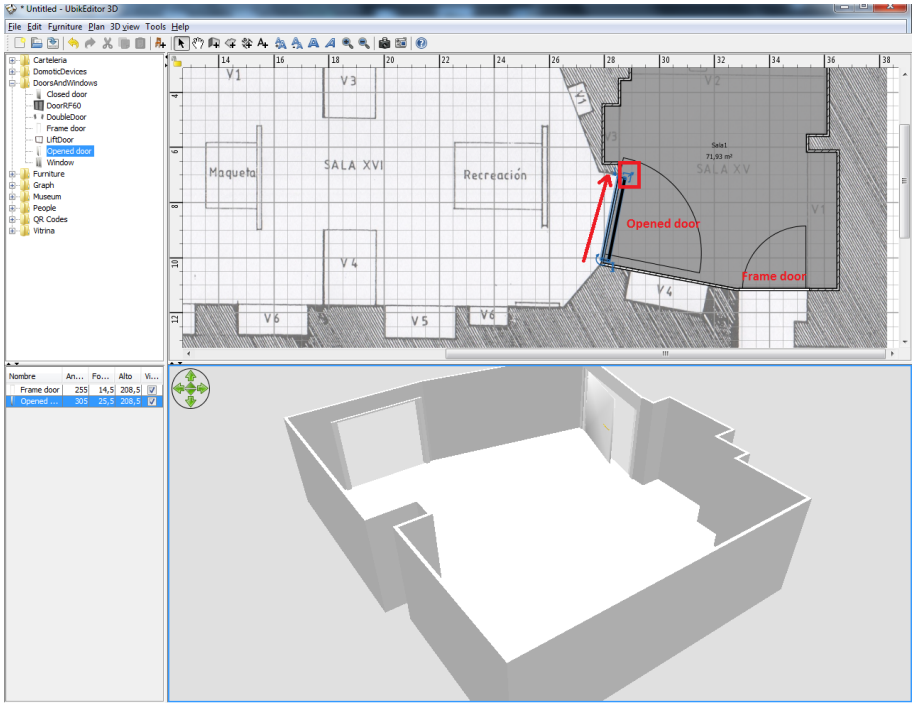
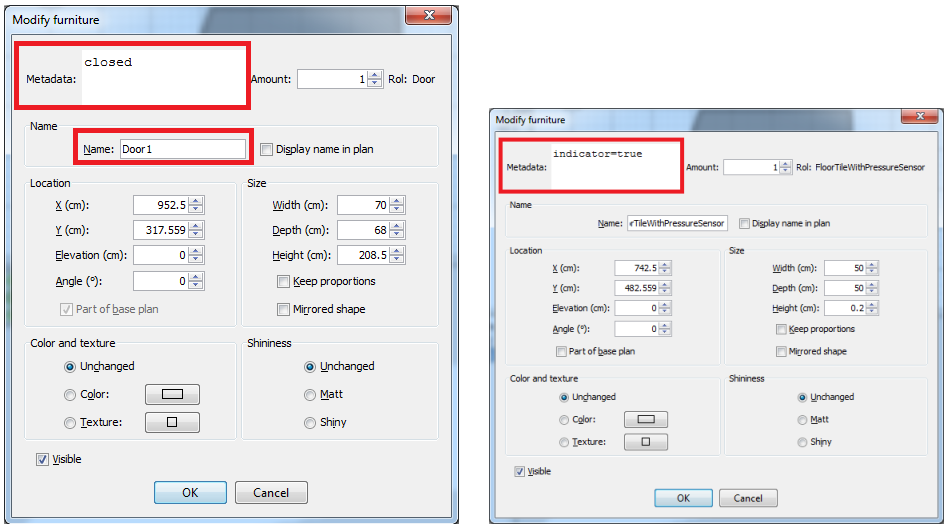


Figure 12: Resizing a door.



(a) (b)

Figure 13: (a) How to edit the door type (opened, closed or locked), (b) Floor tile with press sensor properties

2.2.1 Some notes about the doors and the simulation

- When during a simulation, an agent calculates routes of how to go from one place to another, the routes that cross a door locked will be ignored.
- Doors can be opened, closed or locked during the simulation using the methods: *open()*, *close()*, *lock()* and *unlock()* of the class *sim.app.ubik.building.connectionSpace.Door*.
- The code to obtain the closest door, it is written in a class that heritages from *Person*, an example of use follows:

```
SpaceArea sa = ubik.getBuilding().getFloor(floor).getSpaceAreaHandler().
getSpaceArea(getPosition().x, getPosition().y, Room.class);
Room room = (Room) sa;
ConnectionSpaceInABuilding c =
room.getConnectionSpaceNearerTo(getPosition().x, getPosition().y);
Door door = (Door)c;
door.lock();
door.open();
door.close();
door.unlock();
```

2.3 Door sensor

The door sensor indicates if a door is closed (state *closed* or *locked*) or opened (state *opened*). The door sensor is situated in the objects palette, at the folder *DomoticDevices*. To associate a door with a sensor situate it over the door in the plain. It is very important that the sensor be contained into the door area. This area is the rectangle which illuminates when the door is selected. Figure 11 (b) shows how a sensor is contained into the door area. If it is necessary, resize the sensor and do it lesser.

2.4 Floor tile with pressure sensor

This element is located at the folder DomoticDevices with the name *FloorTileWithPressureSensor*. It consists in a tile of 50x50cm with an associate pressure sensor which detects the weight of persons that step on the floor tile. It is possible to define in the field *metadata* "indicator=true" (figure 13 (b)). In this case, when the tile

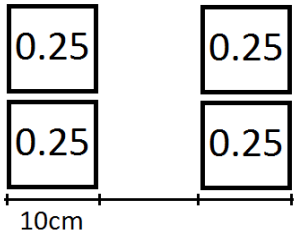


Figure 14: Person weight distribution model.

is pressed during the simulation, it will be coloured in green. If you do not want this feature, you can indicate "indicator=false" instead. To fill a room with tiles of this type it is recommended to select a set of them and (while holding the key *SHIFT*), copy and paste them.

To detect the weight of a person, they must have this property in the field *metadata*. By default, the weight of all the persons in the simulator is 70Kg, represented by the property "weight=70". This weight is distributed in a model as the figure 14 shows. At a distance of 10 cm, the weight is distributed in two spaces of 10cm (simulating the foot). Every space corresponds to a point where the 0.25 of the total weight of the person falls.

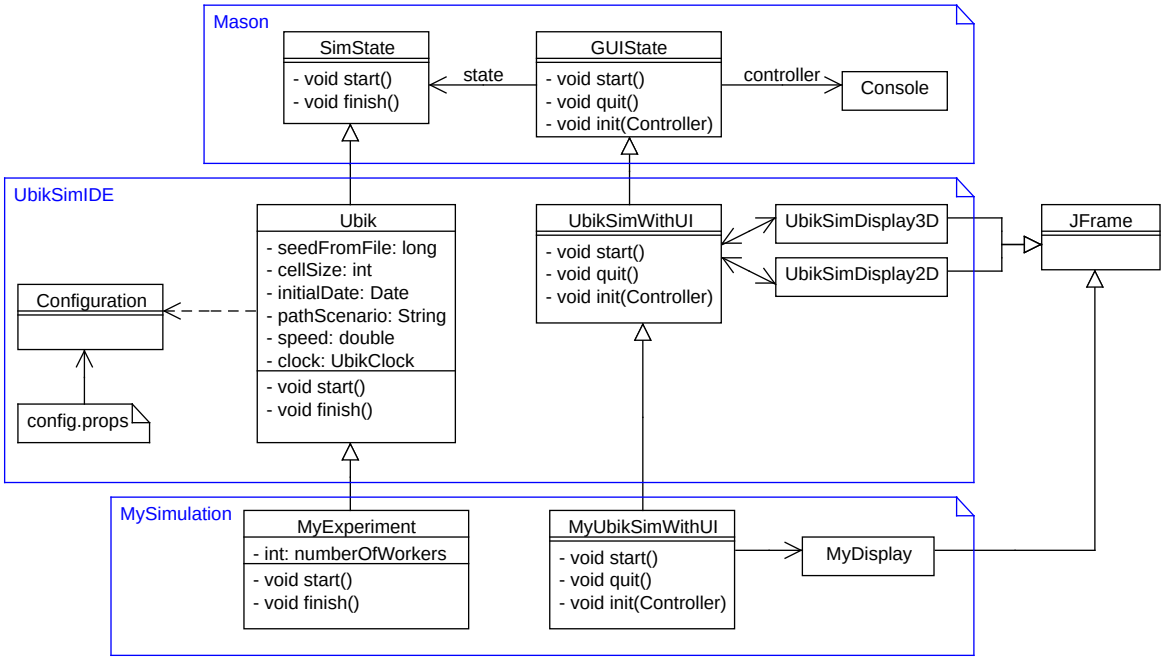


Figure 15: Diagram of the main classes.

3 How to extend UbikSimIDE to develop your own simulation.

This section is an overview of the main classes of UbikSimIDE and it explains how to extend these classes in order to perform our simulations properly.

UbikSimIDE is based on MASON[6], and most of its facilities are available from UbikSimIDE. So, it is recommended to read sections 1 and 2 of the MASON manual[7]. Take in mind that UbikSimIDE visualization (2D and 3D) is not performed using MASON.

Figure 15 shows how UbikSimIDE extends SimState and GUIState. SimState represents a simulation and GUIState is the interface used to control and display the simulation. Ubik extends SimState since it is a basic simulation which has a seed, a cellSize (granularity of space), a initial date of the simulation, a path where the scenario (.ubiksim) is located. All these attributes can be loaded from a file using Configuration class. This class has get and set methods in order to make easier their management. By default, the file is called "config.props" but you can specify any other in the constructor method (new Configuration("file path")). If you do not pass a configuration object to Ubik class, it creates one using "config.props".

Listing 1 shows how to run UbikSim using the Configuration class.

```
1  public static void main(String []args) {
2    Configuration configuration = new Configuration("myconfig.props");
3      configuration.setPathScenario("myScenrio.ubiksim");
4      configuration.setSeed(0123456);
5      configuration.setInitialDate(new Date(System.currentTimeMillis()));
6
7      Ubik ubik = new Ubik(configuration);
8      UbikSimWithUI vid = new UbikSimWithUI(ubik);
9      Console c = new Console(vid);
10     c.setIncrementSeedOnStop(false);
11     c.setVisible(true);
12 }
```

[6]MASON website: `http://cs.gmu.edu/~eclab/projects/mason/`
[7]MASON Manual: `http://cs.gmu.edu/~eclab/projects/mason/manual.pdf`

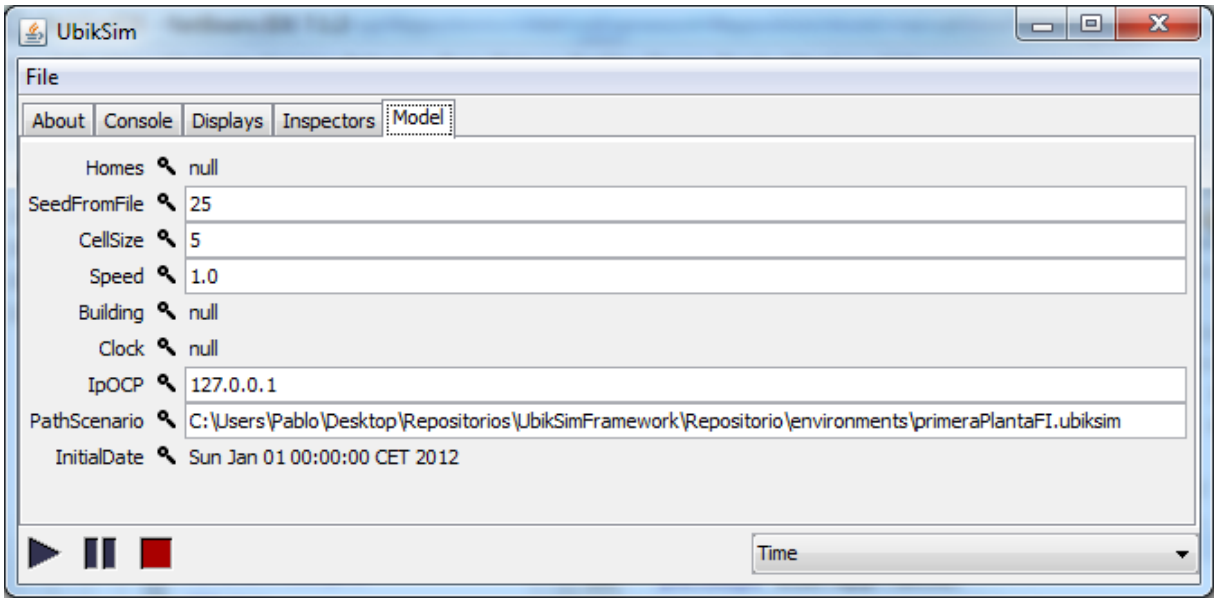


Figure 16: Model tab of the console with simulation parameters.

Listing 1: Usage of the Configuration class.

Ubik class initializes its attributes using configuration class. The initialization is carried out in the constructor method in order to be able to change these attributes from Model tab of the console. It is explained below.

UbikSimWithUI class extends GUIState and offers two displays. UbikSimDisplay3D is used to watch the simulation in 3D and allow you to control a character of the simulation. UbikSimDisplay2D is used to inspect the properties of the entities of the simultation on the Inspector tab. By double-click on an entity in the UbikSimDisplay2D, their properties are shown in the tab. Every display should extend JFrame class.

Next two subsections explain how to extends Ubik and UbikSimWithUI classes to customize simulations.

3.1 MyExperiment

Every experiment (or simulation) must extend Ubik class (such as MyExperiment). This class has attributes related with our simulation. If we create get and set methods for each attribute, they will could be edit in the Model tab of the console, see Figure 16. Fields are not editable since they had not defined their set methods. It is recommended to define a method that has a seed as a parameter and/or one that has a Configuration class as parameter. Both constructors must call to their parent constructor, see listing 2.

The *start()* method should call super method and initialize variables related with the scenario. The scenario is created in Ubik.start() method. It is important, because in the constructor, the environment is not created yet.

In MyExperiment, it duplicates an agent whose name is “Agent” and it is moved to a random position. The agent must exists in the scenario. Then, the name of each agent is changed to “Agent-x”, i.e., “Agent-1”, “Agent-2”, etcetera.

The finish() method should be redefined in order to release resources created in the class. In this case it is not necessary.

```
1  public class MyExperiment extends Ubik {
2
3    int numberOfAgents = 100;
4
5    public MyExperiment(long seed) {
```

```
6       super(seed);
7     }
8
9     @Override
10    public void start() {
11      super.start();
12
13      Automaton.setEcho(false);
14
15      Person pattern = PositionTools.getPerson(this, "Agent");
16          PersonHandler ph = getBuilding().getFloor(0).getPersonHandler();
17          ph.addPersons(getNumberOfAgents(), true, pattern);
18      ph.changeNameOfAgents("Agent");
19    }
20
21    public int getNumberOfAgents() {
22      return numberOfAgents;
23    }
24
25    public void setNumberOfAgents(int numberOfAgents) {
26      this.numberOfAgents = numberOfAgents;
27    }
28  }
```

Listing 2: MyExperiment class

The experiment can be launched without graphical user interface and with total control of each step with the code in listing 3. In the example the experiment is run until step number 5000.

```
1  public static void main(String[] args) {
2    SimState state = new Students(System.currentTimeMillis());
3    state.start();
4    do
5      if (!state.schedule.step(state)) break;
6    while(state.schedule.getSteps() < 5000);
7    state.finish();
8    System.exit(0);
9  }
```

Listing 3: Launching a controlled experiment without graphical user interface.

In order to monitoring the simulation with a graphical interface, the experiment has to be launched using UbikSimWithUI class, see listing 4.

```
1  public static void main(String []args) {
2      Ubik ubik = new MyExperiment(0);
3      UbikSimWithUI vid = new UbikSimWithUI(ubik);
4      Console c = new Console(vid);
5      c.setIncrementSeedOnStop(false);
6      c.setVisible(true);
7  }
```

Listing 4: Launching a controlled experiment with default graphical user interface.

3.2 MyUbikSimWithUI

In order to add windows to control any element of the simulation, such as console to send commands to the agents, UbikSimWithUI class should be extended, such as MyUbikSimWithUI, see Figure 15. UbikSimWithUI already has to displays. 2D display is used to inspect properties of every element of the scenario by double clicking on it. The properties are presented in Inspector tab of the console. 3D display is used to watch the 3D simulation.

The display should extend JFrame class.

Displays must be created and registered to the console in the start() method, see listing 5. Displays can be set visible or not by default. Anyway, displays can be shown or hidden using Displays tab of the console. finish() method should be redefined to unregister every display registered in start() method.

```
1  public class MyUbikSimWithUI extends UbikSimWithUI {
2
3      private MyDisplay myDisplay;
4
5      @Override
6      public void start() {
7          super.start();
8
9          myDisplay = new MyDisplay(this);
10         myDisplay.setVisible(true);
11         controller.registerFrame(ubikSimDisplay2D);
12     }
13
14     @Override
15     public void finish() {
16         super.finish();
17
18         controller.unregisterFrame(myDisplay);
19     }
20 }
```

Listing 5: MyUbikSimWithUI class.

To launch our customized console, UbikSimWithUI is replazed by MYUbikSimWithUI, see listing 6.

```
1  public static void main(String []args) {
2      Ubik ubik = new MyExperiment(0);
3      UbikSimWithUI vid = new MyUbikSimWithUI(ubik);
4      Console c = new Console(vid);
5      c.setIncrementSeedOnStop(false);
6      c.setVisible(true);
7  }
```

Listing 6: Launching the experiment with a customized GUI.

The UbikSimclass project available on-line (visit UbikSim site) model a test person escaping from a building and offer an example of batch experiments in EscapeSimBatch (see Section 5). Hereinafter the code of this project will be used as example. In this project, EscapeSim class (see Listings 7) extends Ubik class and EscapeSimWithGUI (see Listings 8) extends UbikSimWithUI.

```
1  public class EscapeSim extends Ubik {
2
3       static int maxTimeForExecution=1500;
4
5       /**
6        * Object with information about execution and, if needed,
7        * to finish the execution
8        */
9       EscapeMonitorAgent ema ;
10      Fire fire;
11
12     /**
13      * Passing a random seed
14      * @param seed
15      */
16     public EscapeSim(long seed)   {
17         super(seed);
```

```
18
19      }
20
21        /**
22       * Passing a random seed and time to make EscapeMonitorAgent to finish simulation
23       * This time must be less than maxTimeForExecution
24       * @param seed
25       */
26      public EscapeSim(long seed, int timeForSim)   {
27          super(seed);
28          EscapeMonitorAgent.setStepToStop(timeForSim);
29
30      }
31
32
33      /**
34       * Using seed from config.pros file
35       */
36       public EscapeSim() {
37             super();
38             setSeed(getSeedFromFile());
39      }
40
41       /**
42        *
43       * Adding things before running simulation.
44       * Method called after pressing pause (the building variables are instantiated)
             but before executing simulation.
45
46        */
47     public void start() {
48          super.start();
49          ema= new EscapeMonitorAgent(this);
50          fire= new Fire(this);
51          Automaton.setEcho(false);
52          //add more people
53          PersonHandler ph=  getBuilding().getFloor(0).getPersonHandler();
54          //ph.addPersons(20, true, ph.getPersons().get(0));
55          //change their name
56          ph.changeNameOfAgents("a");
57     }
58
59
60     /**
61    * Default execution without GUI. It executed the simulation for maxTimeForExecution
         steps.
62    * @param args
63    */
64       public static void main(String []args) {
65
66          EscapeSim state = new EscapeSim(System.currentTimeMillis());
67          state.start();
68           do{
69                   if (!state.schedule.step(state)) break;
70           }while(state.schedule.getSteps() < maxTimeForExecution);//
71           state.finish();
72
73
74      }
75
```

```
76      /**
77       * Get the Fire object from the simulation object
78       * @return
79       */
80
81      public Fire getFire(){
82          return fire;
83      }
84      /**
85       * Get the monitor agent (agent logging data) from the simulation object
86       * @return
87       */
88    public EscapeMonitorAgent getMonitorAgent(){
89          return ema;
90      }
91
92
93
94
95  }
```

Listing 7: EscapeSim class.

```
1   /**
2    * Start the sim with a GUI, displays show 3d and 2d views
3    * @author Emilio Serrano, Ph.d.; eserrano@gsi.dit.upm.es
4    */
5   public class EscapeSimWithGUI extends UbikSimWithUI {
6
7       public EscapeSimWithGUI(Ubik ubik) {
8       super(ubik);
9     }
10
11      /**
12       * Method called after pressing pause (the building variables are instantiated)
             but before executing simulation.
13       * Any JFrame can be registered to be shown in the display menu
14       */
15      @Override
16      public void start() {
17          super.start();
18          ((EscapeSim)state).fire.insertInDisplay();
19
20      }
21
22  /**
23   * Executing simulation with GUI, it delegates to EscapeSim, simulation without GUI
24   * @param args
25   */
26      public static void main(String []args) {
27          EscapeSim escapesim = new EscapeSim(System.currentTimeMillis());
28          EscapeSimWithGUI vid = new EscapeSimWithGUI(escapesim);
29          Console c = new Console(vid);
30          c.setIncrementSeedOnStop(false);
31          c.setVisible(true);
32      }
33
34
35
36
```

```
37
38
39
40
41
42 }
```

Listing 8: EscapeSimWithGUI class.

3.3 Launching our experiment using UbikSimIDE

An experiment can be launched from the UbikSimIDE. By default, UbikSimIDE launch Ubik class, see method execute of SimulationAction in listing 9. Note that Configuration class is used and the path of the scenario is that which is opened. If we want to launch our experiment by pressing Tools → Simulation form UbikSimIDE, Ubik class has to be replaced by MyExperiment such as was explained before.

```
1  public class UbikSimPlugin extends Plugin {
2
3      private static Plugin plugin;
4
5      @Override
6      public PluginAction[] getActions() {
7          setPlugin(this);
8          // TODO Auto-generated method stub
9          return new PluginAction[]{new SimulationAction()};
10     }
11
12     public static Plugin getPlugin() {
13         return plugin;
14     }
15
16     public static void setPlugin(Plugin plugin) {
17         UbikSimPlugin.plugin = plugin;
18
19     }
20
21     public class SimulationAction extends PluginAction {
22
23         public SimulationAction() {
24             putPropertyValue(Property.NAME, "Simulation");
25             putPropertyValue(Property.MENU, "Tools");
26             // Enables the action by default
27             setEnabled(true);
28         }
29
30         @Override
31         public boolean isEnabled() {
32             return getHome() != null && getHome().getRooms() != null && getHome().
                   getRooms().size() > 0;
33         }
34
35         @Override
36         public void execute() {
37             Configuration configuration = new Configuration();
38             String home = getHome().getName();
39             configuration.setPathScenario(home);
40
41             Ubik ubik = new Ubik(configuration);
42             UbikSimWithUI vid = new UbikSimWithUI(ubik);
43             Console c = new Console(vid);
```

```
44              c.setIncrementSeedOnStop(false);
45              c.setVisible(true);
46          }
47      }
48  }
```

Listing 9: Action to launch simulation form UbikSimIDE.

4 Navigation in the environment

UbikSim offers realistic environments, see section 2, while other simulation tools may offer a simple grid to represent the world. This makes the simulation more descriptive, but also more complex. Specifically, agents have to be able to recover information from the environment and to move through it. The navigation problem consists of making an agent go from one point (x_1, y_1) to another point (x_1, y_1) avoiding obstacles such as walls and other agents.

The UbikSim class project available on-line (visit UbikSim site) models a test person escaping from a building. In this example, as section 6 explains, agents move through a complex environment by simply using a class called *"Move"*. However, if it does not work correctly or according to the requirements of the specific simulation, this section offers insights into the navigation given by UbikSim.

4.1 Navigation without tools

The readers used to other tools such as MASON, NetLogo or Repast; may want to program their navigation by only coordinates in the space. For that purpose, assuming an object ubik which extends the class *Ubik* (which represents a simulation, see section refextending), the following method returns an object *SparseGrid2D* (see MASON documentation) with the space of a floor:

```
1 ubik.getBuilding().getFloor(0).getSpaceAreaHandler().getGrid();
```

However, UbikSim offers more abstract tools for the navigation. Specifically, the environment created by UbikEditor is covered automatically by a number of nodes which form a graph that developers could use for minimum paths algorithms.

4.2 Automatic navigation graph

The class *BuildingToGraph* deals with the construction of a graph when the environment is built. More concretely, its method *createRooms* introduces a number of nodes automatically in each room:

- A node in the center of the room (to move to a room).
- A node for each piece of furniture (to move to them inside a room). The kind of furniture must be registered in the file *"/handlers/furnitures.txt"*.
- A node for each "connection space" in the room, such as doors or stairs. These objects are instances of *ConnectionSpaceInABuilding* and allow the graph to connect different rooms.

Besides, this method, *createRooms*, generates the edges inside a room if there is no obstacle between the nodes (it uses *getSpaceAreaHandler().isObstacles(...)* for this). The edges are weighted with the distance between the nodes connected.

After creating the navigation graph for each room, rooms have to be connected. For that purpose, *BuildingToGraph* calls the method *createConnectionSpace* which connect edges between the nodes created for doors and stairs (objects of *ConnectionSpaceInABuilding*) in different rooms. The weight given to these edges is stated by *ConnectionSpaceWeighted* to give, for example, an infinite weight for close doors.

Warning!. A very important note is that the environment should include "node graphs" manually introduced by the user in UbikEditor (see section 2) for large rooms such as corridors (a node graph in front of each door is recommended) or rooms which non-rectangular shapes (for example, a room with L-shape should include a node graph at the corner).

At this point, the developer can choose to use the graph generated by UbikSim directly (possibly after modifying its generation, for example to add name for special nodes considered in the simulation). This graph is available in the graph attribute of the class RoadMap. On the other hand, next section gives methods which automatically obtain paths from the graph.

```
1 ubik.getBuilding().getBuildingToGraph().getRoadMap().getGraph()
```

4.3 Navigating using the graph

The class *RoadMap* takes the graph generated in the above section and uses Dijkstra's algorithm provides several methods to obtain a path from two points in the space and returning it as (1) a list of graph nodes (*getGraphPath*); or, (2) a list of positions of the grid (*getRoute*).

```
1 ubik.getBuilding().getBuildingToGraph().getRoadMap().getGraphPath(Int2D ori, Int2D
      dest)
```

Note that these methods stored the routes previously calculated, so if the environment is supposed to change dynamically (for example, doors are opened and closed during the execution) this feature should be removed. The method *getGraphPath* checks these routes already calculated in the matrix *routes[p][q]*.

5 Batch experiments, logging data, and conducting statistic operations

When working with UbikSim, developers probably are interested in executing not only one simulation but hundreds of them to learn about the model. For that purpose, batch experiments must be conducted. **The ubikSim class project available on-line (visit UbikSim site) models a test person escaping from a building and offers an example of batch experiments in EscapeSimBatch. Hereinafter the code of this project will be used as example**.

A batch class must:

- Iterates over the different parameters of the simulation experimentsForDifferentParameters()
- Executes a batch of experiments for those parameters batchOfExperiments()
- Executes one experiment for each element of the batch oneExperiment(int seed). The series of seeds must be repeated for each different combination of parameters.
- Performs statistical operations on results and prints them in files. The example assumes 2 operations (in 2 files): mean and standard deviation for the metrics logged. It can be easily extended with more operations.

The EscapeSimBatch example is shown below:

```
1  /**
2   *
3   * Batch of experiments for EscapeSim
4   * @author Emilio Serrano, Ph.d.; eserrano@gsi.dit.upm.es
5   */
6  public class EscapeSimBatch {
7      static int experimentsPerParameters = 3;
8      /**
9       * Time for each experiment, EscapeMonitorAgent will end simulation at this step
10      */
11     static int timeForExperiment = 1000;
12     /**
13      * Extra heading added for each parameters configuration in the data logged by
            EscapeMonitorAgent
14      */
15     static ArrayList<String> extraHeadingsPerParameter;
16     /**
17      * name with output
18      */
19     static String fileName;
20
21   public static void main(String[] args) throws IOException {
22       //name of the batch file
23        String date = (new Date()).toString().replace(':', '.');
24        fileName = "Batchoutput␣" + date;
25        ArrayList<GenericLogger> r = experimentsForDifferentParameters();
26          printInFile(r);
27          System.exit(0);
28
29 }
30
31   /**
32    * Experiments for different parameters of the configuration, example: number of
          agents
33    * @return
34    * @throws IOException
35    */
```

```
36      private static ArrayList<GenericLogger> experimentsForDifferentParameters() throws
            IOException {
37          ArrayList<GenericLogger> r = new ArrayList<GenericLogger>();
38          extraHeadingsPerParameter = new ArrayList<String>();
39          r.addAll(batchOfExperiments());
40          /*if needed, add a loop with an interation for parameter,  change the
                simulation parameters,
41           * and add an extra heading per configuration to distinguish the different
                 batch in the output file
42           */
43          extraHeadingsPerParameter.add("");//no extra heading for mean
44          extraHeadingsPerParameter.add("");//no extra heading for deviation
45          deleteTempFiles();
46
47          return r;
48      }
49
50         /**
51        * A batch of experiments.
52        * Mean and deviation of experiments are registered in the list of result
53        *
54        * @return generic loggers with the results registered (mean and deviation)
55        */
56       private static ArrayList<GenericLogger> batchOfExperiments() {
57          ArrayList<GenericLogger> listOfResults = new ArrayList<GenericLogger>();
58          for (int i = 0; i < experimentsPerParameters; i++) {
59              GenericLogger gl1 = oneExperiment(i * 1000);//seed shoud be equal for
                    different parameters
60              listOfResults.add(gl1);
61          }
62          ArrayList<GenericLogger> r = new ArrayList<GenericLogger>();
63          r.add(GenericLogger.getMean(listOfResults));
64          r.add(GenericLogger.getStandardDeviation(listOfResults));
65          return r;
66
67       }
68         /**
69        * A simple experiment , code based on the main method of EscapeSim
70        * @param seed
71        * @return
72        */
73       public static GenericLogger oneExperiment(int seed) {
74
75          EscapeSim state = new EscapeSim(seed,timeForExperiment );
76          state.start();
77          do{
78                  if (!state.schedule.step(state)) break;
79          }while(state.schedule.getSteps() < timeForExperiment*2);//the
                EscapeMonitorAgent will finish before
80          state.finish();
81          return state.ema.getGenericLogger();
82       }
83
84  /**
85   * Print results from the generic logger of each execution assuming that mean and
         deviation has been logged
86   * @param r
87   * @throws IOException
88   */
89      private static void printInFile(ArrayList<GenericLogger> r) throws IOException {
```

```
90
91  ...
```

As seen, batch experiments in this example deal with *genericLogger* objects which are filled by an *EscapeMonitorAgent*. This is a logger agent which is created in the start method of the simulation without GUI (so it is also present in batches and experiments with GUIs). Logger agents are very simple MASON agents which extends from MonitorService, requiring to define:

- register(): To be included in the schedule.
- step(SimState ss): To perform actions which basically are: (1) fill a generic logger (or your favorite data structure), (2) check if the simulation has to finish to kill it.

The EscapeMonitorAgent code is given below:

```
1   public class EscapeMonitorAgent implements MonitorService {
2   /**This agent stops simulation in this step   */
3       protected static long maxStepToStop=1000;
4
5       /**
6        * Data structure to log data and conduct statistical operations
7        */
8       protected GenericLogger genericLogger;
9       protected Ubik ubik;
10      /**
11       * Counter with people who have touch fire
12       */
13      protected int  peopleCloseToFire=0;
14
15
16
17      public EscapeMonitorAgent(Ubik u) {
18          this.ubik = u;
19          String logHeadings[]={"PeopleInBuilding","PeopleWhoReachFire"};
20          genericLogger = new GenericLogger(logHeadings);
21          register();
22      }
23
24      /**
25       * Set the number of the step to stop simulation
26       * @param s
27       */
28      public static void setStepToStop(long s){
29          maxStepToStop=s;
30      }
31      /**
32       * Get the genericLogger with the data of the simulation
33       * @return
34       */
35      public  GenericLogger  getGenericLogger(){
36         return genericLogger;
37      }
38
39   /**
40    * Register the fire in the schedule (to make the simulation call the step method in
          fire for each step).
41    */
42
43      public void register() {
44          ubik.schedule.scheduleRepeating(this, 1);
45      }
```

```
46  /**
47   * Method with the actions to be performed by this agent in each step: adding data in
         genericLogger and checking if the simulation has to finish
48   *
49   * @param ss
50   */
51      public void step(SimState ss) {
52          List<Person> people = ubik.getBuilding().getFloor(0).getPersonHandler().
                getPersons();
53          double[] toLogInStep ={ people.size(), peopleCloseToFire};
54          genericLogger.addStep(toLogInStep);
55
56          if(ubik.schedule.getSteps()>=maxStepToStop){//end simulation from agent
                monitor
57              ubik.kill();
58          }
59
60      }
61
62      public void stop() {
63          throw new UnsupportedOperationException("Not␣supported␣yet.");
64      }
65
66
67
68  }
```

6 Modelling people in UbikSim

The example of person given in the simulator is TestPerson. It extends the Person class and has a method step as any MASON agent to implement its behaviour.

6.1 Creating crowds in the simulator

The simplest way to model a group of people is using the field "amount" of a person inserted from the editor. Putting a value of x, x copies of the aforementioned agent are distributed at random positions in the environment. Only positions in rooms with name are considered.

Groups of agents can also be created and deleted dynamically from the code (even during a simulation). For that purpose, methods of the *PersonHandler* class which are useful are: *addPersons* and *removePersons*. Another useful method is *changeNameOfAgents* that gives each agent x a name "a_x". These methods are usually called in the method start of the simulation without GUI extending the Ubik class.

Sometimes, it is also interesting to model agents who exit of the graphical representation without occupying any space or losing their execution turn, for example when modelling a room where agents can come in and out without modelling what is outside of the room. Subsequently, this agent must be introduced again in the environment. For these cases, the *PositionTools* class provides developers with the following methods: *isInSpace*, *putInSpace*, and *getOutOfSpace*.

6.1.1 Recovering the initial position of persons added in runtime

The following code can be used in the step method of a person to make sure that the initial position is available when needed:

```
1  if(initialPosition==null) {
2    SpaceArea sa = (SpaceArea) ubik.getBuilding().getFloor(0).getSpaceAreaHandler().
         getSpaceArea(position.x, position.y);
3      if(sa instanceof Room) {
4        initialPosition = (Room) sa;
5      } else if(sa instanceof Door) {
6        Door door = (Door) sa;
7        Int2D point = door.getAccessPoints()[0];
8        initialPosition = (Room) ubik.getBuilding().getFloor(0).getSpaceAreaHandler().
             getSpaceArea(point.x,point.y);
9      }
10 }
```

6.2 Using the Automaton class

6.2.1 Introduction to Automaton

As said, there is a step method in TestPerson (and Person) to implement any behaviour. However, UbikSim gives a top-down approach for modelling complex agents by using the class Automaton. This approach involves the breaking down of a human behaviour to gain insight into its compositional sub-behaviours.

Automaton is a hierarchical automaton of behaviours (HAB). Each state is another automaton (with sub states which also are automata). There is a main automaton (the one connected directly with the Person), father automata and subordinate automata. Maybe you see it better as a tree of behaviours. Figure 17 offers an example to illustrate the idea of Automaton. A baby delegates to babyAutomaton (extending Automaton) to implement baby's behaviour. This automaton can generate transitions to the states: play, be hungry, sleep, and so on. Some of these states are complex enough to be further detailed. Actually these states are also instances of Automaton, so they have more subordinate automaton/states (Play can generate transitions to use toy and break toy). The idea of the tree is useful, but it is not a tree (break toy can delegate to cry). **The UbikSim class project available on-line (visit UbikSim site) models a test person escaping from a building with the classes described in figure 18. Hereinafter the code of this project will be used as example**. The following code shows the person creating and delegating to the automaton:

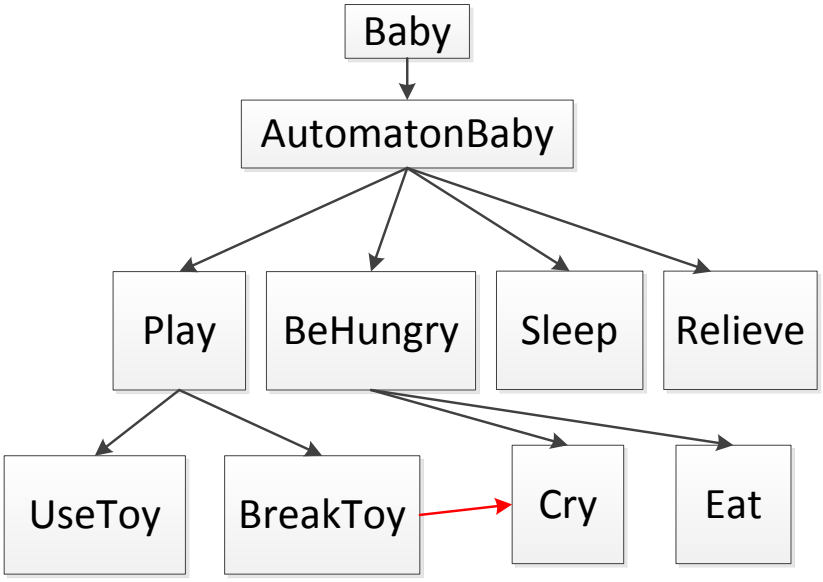


Figure 17: Classes with the behaviour of a baby

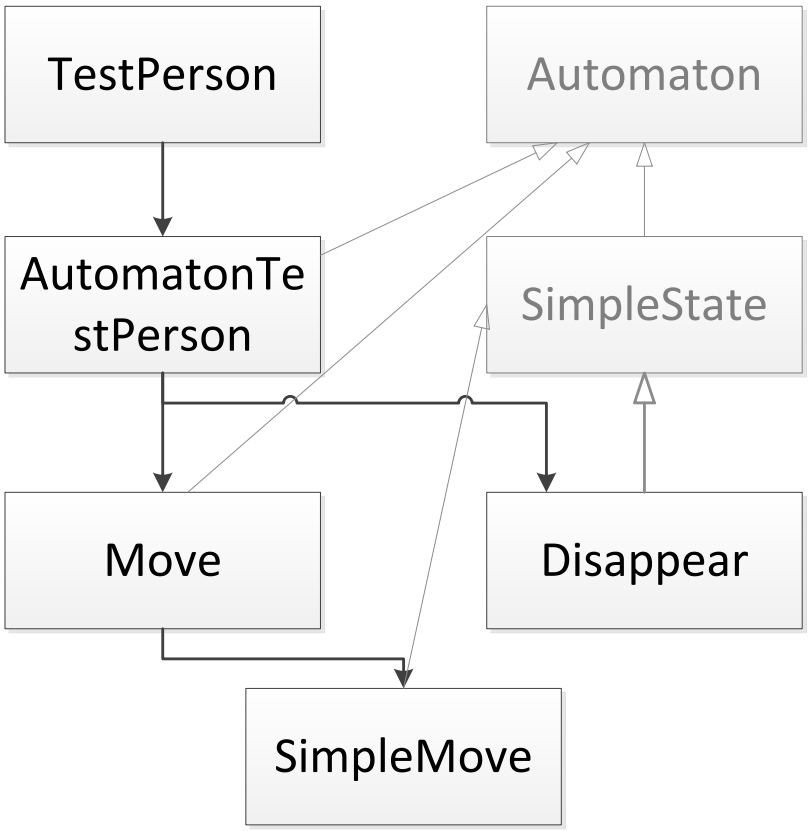


Figure 18: Classes with the behaviour of a test person in the ubikSim class project

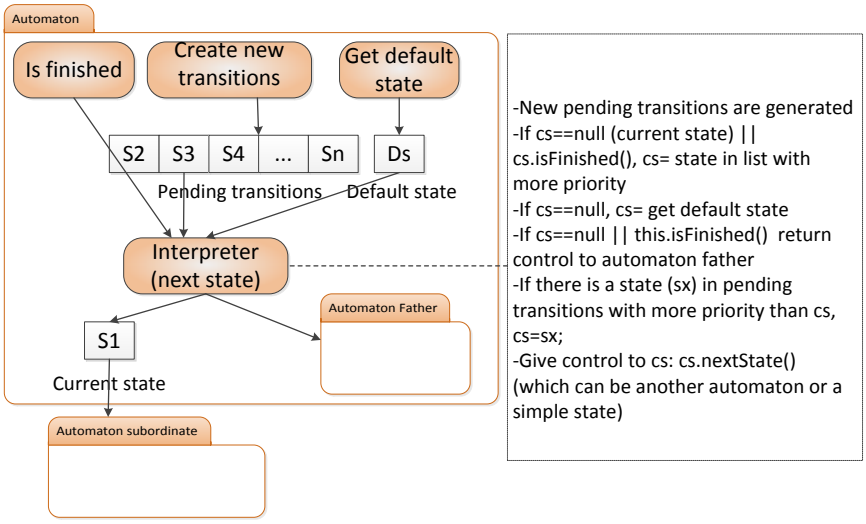


Figure 19: Automaton components and interpreter

```
1  public void step(SimState state) {
2  ...
3    if(automaton==null) {
4      automaton = new sim.app.ubik.behaviors.escape.AutomatonTestPerson(this);
5    }
6    automaton.nextState(state);
7  }
```

How does automaton decides what transition is taken in each moment, when to delegate to a subordinate automaton/state, when to return control to the father automaton? Automaton has an interpreter which decides these actions based on the implementation of several methods that developers have to fill when an instance of Automaton is created. Figure 19 illustrates the components of an Automaton. In essence, an Automaton is composed of: a list of pending transitions ordered by priority, a method for creating new transitions (adding them to the list), a current state (state with the control) and a default state (state that takes control of the automaton when the list of pending transitions is empty).

An *interpreter* [**?**] makes the automaton advance every step of the simulation. The main actions of the interpreter are described below. (1) Firstly, *new pending transitions* are generated from the *current state* to another. (2) Then, the interpreter checks whether the current state is completed (or still empty) to undertake the next transition in the list, the one with higher priority. (3) If the list is empty, the *default state* is taken as current state. (4) If the current state is not finished but there is a transition with highest priority in the list, the interpreter gives the control to the latter. Finally, (5) the current state takes control to undertake an action. The Automaton control is performed transparently to the developer. The complete code of the interpreter is in the GitHub repository of UbikSim (nextState method in Automaton) and shown here:

```
1  public void nextState(SimState simState) {
2     //ignore if the state is paused
3     if(pause) return;
4     //generating new transitions and include them in the pending transition list
5     ArrayList<Automaton> newTransitions = createNewTransitions(simState);
6     if(newTransitions!=null && !newTransitions.isEmpty()){
7       for( Automaton ps: newTransitions){
8         this.addTransition(ps);
9       }
10    }
11    //check if current state is finished and make transition or returning control to
```

```
        automaton father
12      if (currentState== null || currentState.isFinished(simState)) {
13          if(currentState!=null){//current state is finished
14           currentState.setFinished(true);//set it as finished
15          }
16          if(!pendingTransitions.isEmpty()){//next pending transition
17           currentState=getTransitionAccordingToPriority(null, simState);
18          }
19          else{//go to default state
20          currentState = getDefaultState(simState);
21           }
22          if(currentState==null){//returning control to automaton father (no pending
                transitions or default state)
23           this.setFinished(true);
24         return;
25          }
26      }
27      //stop current state to start make a transition to a state with more priority
28    if ((!pendingTransitions.isEmpty()) && pendingTransitions.getFirst().priority >
          currentState.priority) {
29          currentState =  getTransitionAccordingToPriority(currentState,simState);
30      }
31      //give control to subordinate automaton
32      currentState.nextState(simState);
33      //reducing time for the current state
34      if (currentState.duration != -1) {
35          currentState.decreaseTimeLeft();
36      }
37  }
```

Note that the interpreter assumes that an automaton ends and returns the control when there is no default state given and no transitions pending.

Very important, if you do not like or understand Automaton, forget it and just use the step in person to implement your behaviour as in MASON!.

6.2.2 Creating your Automaton instance

The first thing to decide is if it is a main automaton (a person delegates to it), a subordinate automata (another automaton delegates to it) or a simple state (it does not have to delegate to other automata because it is very simple). The first two cases are instances of Automaton and they basically differ in:

- The constructor. A main automaton does not have to compete against other automata, so they do not need priority. The following code shows the constructor of a main Automaton for Test Person.

```
1  public AutomatonTestPerson(Person personImplementingAutomaton) {
2    super(personImplementingAutomaton);
3  }
```

  And the following code shows the constructor for a subordinate automaton which needs a priority, a name, a maximum time given to execute the behaviour, and extra information for the behaviour (such as a destiny if it is a move behaviour):

```
1  public MoveToClosestExit(Person personImplementingAutomaton, int priority, int
       duration, String name){
2    super(personImplementingAutomaton,priority, duration, name);
3  }
```

- The final condition. Note also that the main automaton (level 0), the highest in the hierarchy, never ends if a cyclic behaviour wants to be modelled (and therefore a default state must be given). Besides, no implementation is given for isFinished and no maximum time is given in the constructor.

Explained these differences between a main automaton and a subordinate automata, the tasks to be implemented by the developer to create an Automaton are: (1) creating new transitions, (2) getting the default state, and (3) including an ending condition for the automaton if needed.

The following code shows the methods (1) and (2) for the main automaton TestPersonAutomaton:

```
1
2  public class AutomatonTestPerson extends Automaton {
3  ...
4  @Override
5  public Automaton getDefaultState(SimState simState) {
6    return new DoNothing(p,0,1,"doNothing");
7  }
8
9
10 @Override
11 public ArrayList<Automaton> createNewTransitions(SimState simState) {
12   ArrayList<Automaton> r=null;
13   if(!this.isTransitionPlanned("goToExit")){
14     r=new ArrayList<Automaton>();
15     Room exit= PositionTools.closestRoom(p, STAIRS);
16     r.add(new Move(p,10,-1,"goToExit",exit));
17     r.add(new Disappear(p, "␣escaped␣", p.getName() + "␣escaped␣using␣" + exit.
           getName()));
18   }
19
20
21   if(!p.hasBeenCloseToFire()){
22   if (sim.getFire().tauchingFire(personImplementingAutomaton)){
23   p.setCloseToFire(true);
24   sim.getMonitorAgent().peopleCloseToFire++;
25   }
26   }
27
28   return r;
29 }
```

The code returns to the default state DoNothing for one step if the pending transitions are empty. In createNewTransitions, if the agent has not planned to go to an exit (this is done to avoid planning things which are already planned, in other words, to avoid that the pending transition list grows in each step) a transition is planned to go to the closest exit (priority 10) and another to make the agent disappear after reaching the exit.

Sometimes, an automaton must include extra actions at each step besides passing the control to the next level. In that case, those actions or behaviours should be implemented in the methods to generate the *new transitions* or the *default state*. The generation of new transitions is called at each step while obtaining the default state occurs only when the list of transitions is empty. Some typical behaviours included in automatons are: checking that the agent has a specific identification to generate a transition only for that agent (when the same Automaton is used for a number of agents), using a probability distribution function to generate a probabilistic transition, checking the content of the list of pending transitions, checking a flag to generate transitions only once, etc. In the example shown above, some treatment has been included in createNewTransitions: checking if the person is close to a fire.

The following code shows the methods (1) and (2) for the subordinate automaton MoveToClosestExit:

```
1  public class MoveToClosestExit extends Automaton {
2  ...
3   @Override
4  public Automaton getDefaultState(SimState ss) {
5    return null;
6  }
7
8  @Override
9  public ArrayList<Automaton> createNewTransitions(SimState ss) {
```

```
10    <Automaton> r=null;
11     if(!this.isTransitionPlanned("goToExit")){
12      r=new ArrayList<Automaton>();
13      Room exit= PositionTools.closestRoom(p, STAIRS);
14      r.add(new Move(p,10,-1,"goToExit",exit));
15      r.add(new Disappear(p, "␣escaped␣", p.getName() + "␣escaped␣using␣" + exit.
            getName()));
16    }
17    return r;
18  }
```

No default state is given to finish when transitions are performed. In this case, isFinished has not be redefined, but in Move (for example) it is extended to tell the automaton to finish if the destiny has been reached (besides if no more transitions are planned and the time given in the constructor is over).

Explained main automata and subordinate automata, the states at the bottom of the hierarchy (with no subordinate automata) must be explained. These automata implement simple behaviours. The developer must create one of these simple states when defining an action which will be performed in one step to immediately return control to the automaton father. The class that defines simple states in UbikSim is *SimpleState*. (1), (2), and (3), needed in the definition of Automaton instances, are not necessary for simple states because they do not have: transitions, a default state, or a final condition. These states requires redefining only a method *nextState* which carries out an "atomic" action (in the sense that modelling an automaton is not necessary due to its simplicity).

Disappear is an example of such a simple state:

```
1  public class Disappear extends SimpleState {
2  ...
3      public void nextState(SimState state) {
4      if(!personImplementingAutomaton.isStopped()) {
5      if(message!=null) System.out.println(message);
6      personImplementingAutomaton.stop();
7    }
8  }
```

6.2.3 Debugging your Automaton

The method Automaton.setEcho(true) makes automata to print information about the transitions undertaken. Here is an example of the ubikSim class project available on-line. If no output is shown, your agent probably is not connected properly to the automaton defining its behaviour.

```
1  a1, MAINAUTOMATON pending transitions extended [goToExit,  escaped ]
2
3  a1, MAINAUTOMATON automaton changes to state goToExit
4
5  a1, goToExit pending transitions extended [SimpleMove, simple move to (698,157) [
       editor format: 3490,785], SimpleMove, simple move to (727,102) [editor format:
       3635,510], SimpleMove, simple move to (731,99) [editor format: 3655,495],
       SimpleMove, simple move to (744,63) [editor format: 3720,315], SimpleMove, simple
        move to (744,63) [editor format: 3720,315]]
6
7  a1, goToExit automaton changes to state SimpleMove, simple move to (698,157) [editor
       format: 3490,785]
8
9  a1, goToExit automaton finished SimpleMove, simple move to (698,157) [editor format:
       3490,785]
10
11 a1, goToExit automaton changes to state SimpleMove, simple move to (727,102) [editor
       format: 3635,510]
12
13 . . .
```

```
14
15  a1, MAINAUTOMATON automaton finished goToExit
16
17  a1, MAINAUTOMATON automaton changes to state  escaped
```

7 GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.