POLITECNICO

MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Formal Analysis of Search-and-Rescue Scenarios

Project for Formal Methods for Concurrent and Realtime Systems course

Authors: **Alberto Nidasio, Federico Mandelli, Niccolò Betto**

Abstract

This document presents a formal model implemented with Uppaal of search-and-rescue scenarios. Inside a rectangular map of arbitrary size, civilian *survivors* have to be brought to safety by either reaching an exit or being assisted by *first-responders*. *Drones* survey the area and coordinate the rescue efforts by instructing *survivors* on what to do. The model then undergoes formal verification to highlight key behavioral aspects and identify optimal configurations for maximizing *survivors'* safety.

Contents

1 High-Level Model Description

The model adopted for the search-and-rescue mission involves 3 different types of agents: *survivors*, *first-responders* and *drones*. They are placed in different numbers inside a rectangular map, where exits (i.e. safe zones reached by *survivors* to get to safety) and fires are fixed in place from the beginning of the scenario.

The key characteristics of the agents are these:
- **Survivors**: Can be in 3 different states, depending whether they find themselves near a fire or if they are following instructions.
  - **In-need** (i.e. near a fire): They cannot move and needs to be assisted. After T_v time units, they became a casualty.
  - **Busy** (acting as *zero-responders*): The *survivor* is following an instruction and can be either assisting directly or contacting a *first-responder* to get help.
  - **Moving**: When *survivors* are not near a fire or busy enacting some instruction, they can move towards an exit to get to safety following some *moving policy*.
- **First-responders**:
  - **Assisting**: When a survivor *in-need* is within a 1-cell range, the *first-responder* will assist them for T_{fr} time units. After that, the assisted *survivor* is considered safe.
  - **Moving**: When free from other tasks, the *first-responder* can move following some *moving policy*.
- **Drones**: They survey their surroundings, limited by the field of view N_v of the sensors, and following a pre-determined path moving 1 cell at each time step. When two *survivors*, one *in-need* and one free, are detected, the *drone* can instruct the free *survivor* to assist the one *in-need* directly or to contact a *first-responder*.

1.1 Model Assumptions

To simplify the model described in the assignment, the following assumptions have been made:
- The map is a 2D grid with a fixed number of rows and columns, and fires and exits are static (i.e. they won't change during simulation).
- Movements from one cell to another allows for diagonal movements. Therefore, the distance can be easily computed as the maximum between the difference of the x and y coordinates.
- Movements of *survivors* and *first-responders* towards a human target (e.g. a *survivor* goes to a *first-responder*), are modeled with a wait state, where the agents remains idle for the duration of the movement, and then with a change in coordinates. This reduces the complexity of the model, lowering the number of states of the simulation and thus speeding up the verification process.
- *Drones* know the global position of all the *first-responders* and their status, at any given time. This allows them to instruct *survivors* to contact the nearest *first-responder*.
- All *survivors* know the location of the exits and can determine the nearest one.
- *Survivors* cannot start the simulation inside a fire cell.

2 Model Description and Design Choices

2.1 Faster Model

After developing the model described below we discovered that the verification process was too slow, making it difficult to test different scenarios and configurations. To speed up the verification process, we have made some changes to the model described in the following sections. Instead of modeling the interaction between the actors, as they happen in real life, we model only the necessary part of it. Instead of having the actors move on the map, we model the movement as a wait state. This means that when an agent is instructed to move to a certain position, it will wait for a specific amount of time before teleport to the target position. If a movement is composed of different steps (i.e. *zero-responder* going to call a *fist-responder* and then back to the *in-need*) it is now modelled as a unique wait state of the time needed to travel that distance. This simplifies the model and reduces the number of states and transitions, which in turn speeds up the verification process.

When a *drone* detects a survivor *in-need* and a *zero-responder* (and a *first-responder* if needed), it sends a message to the *zero-responder* with the correct wait time (depending on the distance and the assistance time). The same applies to the *first-responder* (if needed) and the *in-need*, so that the *in-need* waits either until they are dead or the wait time has expired (becoming safe).

To reduce computation frequently used variables are stored in global variables, i.e. instead of searching for the position of an actor in the map we store it in a global variable.

In each of the following sections we will describe the model as it was before the changes, and then we will describe the changes made to the model.

2.2 State and Parameters Representation

Each agent type (*survivor*, *first-responder*, *drone*) is represented by an automaton, called **template** in Uppaal. These templates are characterized by many different parameters, and are implemented in Uppaal in the following way:

- The template signature (the parameters list) contains only one constant parameter, the agent id, annotated with a custom type defined as an integer with the range of possible ids (e.g. `typedef int[0, N_DRONES-1] drone_t;`). This way, by listing the template names in the *System declarations* (`system Drone, Survivor, FirstResponder;`), Uppaal can automatically generate the right number of instances of each template;
- The other agents' parameters (e.g. N_v, N_r, T_{zr}, etc.) are defined in constant global arrays (e.g. `const int N_v[drone_t] = {1, 1};`). Each template instance can then index these arrays with its own id to access its own parameters (e.g. `N_v[id]`).

This setup allows for easily defining the simulation parameters all inside the *Declarations* section, thus without modifying either the templates or the *System declarations*, and to easily assign different parameters to each template instance.

In the **faster model** some of the agents parameters are stored in variables, not arrays, loosing the possibility to specify different parameters for each agent but reducing the complexity of the drone template.

2.2.1 Map Representation

Despite each agent holding internally its own position, a global representation of the map is needed for agents who require to know the state of other agents (e.g. *drones* need to know the position of *first-responders* to instruct *survivors* to contact them).

```
// Map cell status enumeration
const int CELL_EMPTY =      0;
const int CELL_FIRE =       1;
const int CELL_EXIT =       2;
const int CELL_FIRST_RESP = 3;
const int CELL_SURVIVOR =   4;
const int CELL_ZERO_RESP =  5;
const int CELL_IN_NEED =    6;
const int CELL_ASSISTED =   7;
const int CELL_ASSISTING =  8;

typedef int[0, 8] cell_t;

// Map array
cell_t map[N_COLS][N_ROWS];
```

The map is represented as a 2D grid of cells, with each cell indicating which type of human agent is within. This choice is made to avoid each agent holding a reference to all other agents, which would make the model more complex and harder to maintain.

When one agent changes position, it updates the map accordingly. For example, when a *survivor* moves, it empties the cell it was occupying and fills the new cell with its type.

```
void move(int i, int j) {
  set_map(pos, CELL_EMPTY);
  pos.x += i;
  pos.y += j;
  set_map(pos, CELL_SURVIVOR);
}
```

In the **faster model** other then the map the position of each actor (and exit) is stored in an array and updated at each movement to reduce calculation in the following templates.
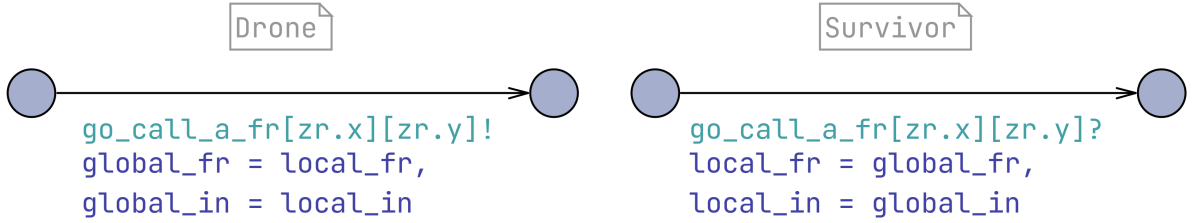
2.3 Synchronization and Message Passing

Between the different agents, there are some interactions that require a way to pass a payload. For example, when a *survivor* is instructed by a *drone* to contact a *first-responder*, the *survivor* must receive from the *drone* both the positions of the *first-responder* and that of the one *in-need* to assist.

This synchronization with message passing is implemented via Uppaal's built-in channels, augmented with global variables used to temporarily store the payload. The channels themselves are 2D matrices to allow targeting a specific agent. Each agent can trigger a channel at a given coordinate or listen to a channel at its own current position.

Following the previous example, the synchronization follows these steps:

1. The *drone* saves the positions of the target *first-responder* and *in-need* in two global variables and then triggers the channel at the coordinates of the targeted *survivor*.
2. The *survivor*, upon receiving the signal through the channel, reads the global variables to get the positions of the *first-responder* and *in-need*.


```
Drone                                    Survivor

go_call_a_fr[zr.x][zr.y]!                go_call_a_fr[zr.x][zr.y]?
global_fr = local_fr,                    local_fr = global_fr,
global_in = local_in                     local_in = global_in
```

In the **faster model** the message passing is simplified by only passing, to all the actors involved, the total waiting time needed to complete the action. This could be a drone passing the waiting time to a *zero-responder* , *in-need* and *first-responder* (if present) or a *first-responder* passing the waiting time to the *in-need*.

2.4 Moving Policies

Both *survivors* and *first-responders* are characterized by a custom moving policy. *Survivors* use this moving policy to reach the nearest exit, while *first-responders* use it to reach a *survivor* in need of assistance.

These moving policies all function in the same way; given a target position, they produce the next move to take towards the target. Therefore, they can be abstracted and implemented in a generalized way.

In order to support random choices, the moving policy implementation works as follows:

- On the edge where we want to perform the move, a non-deterministic selection of offsets i and j is performed to identify a possible adjacent cell to move to (shown in Figure 1);
- The function is_move_valid(i, j) evaluates whether a given adjacent cell is a valid move or not, using the selected moving policy.

To experiment with different moving policies, we have implemented 2 simple policies:

- The **random** policy simply checks if the move is feasible (i.e. whether the cell is not occupied by a fire or another agent). In Figure 2 we can see that the move is valid if the cell is empty. By "enabling" all the feasible moves, the model non-deterministically selects one of them;
- The **direct** policy enables only the moves with the lowest direct distance to the target. As shown by Figure 3, if more than one adjacent cell has the same distance to the target, the policy enables all of them and the model will randomly select among those.

```
i : int[-1, 1],
j : int[-1, 1]
t ≥ 1 &&
is_move_valid(i, j)
move(i, j),
t = 0

        Available
        t ≤ 1
```
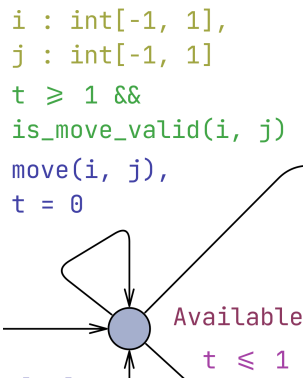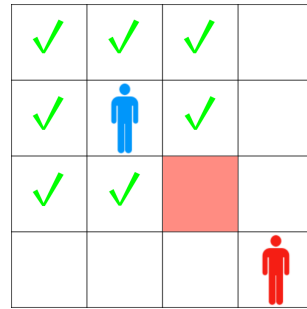
Figure 1: Moving edge
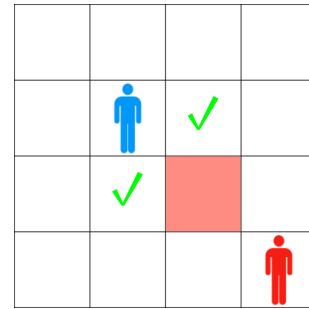
Figure 2: Moving policy RANDOM

Figure 3: Moving policy DIRECT

Below are the implementations of the two moving policies. The *direct* policy computes the distance of the best feasible move to the target, and then enables all those moves with that same distance. The policy is implemented this way in order to avoid preferring one direction over another if more than one move has the same distance to the target.

```
// Random policy allows all movements that are feasible
bool random_is_move_valid(pos_t pos, pos_t move, pos_t target, cell_t type)
{
    return is_move_feasible(pos, move, type);
}


// Direct policy follows the best direct path
bool direct_is_move_valid(pos_t pos, pos_t move, pos_t target, cell_t type)
{
    int min_distance;

    if (!is_move_feasible(pos, move, type))
        return false;

    // Find the distance to the target of the best possible move
    min_distance = compute_best_move_distance(pos, target, type);

    // A move to be valid must have minimum distance
    return distance(move, target) == min_distance;
}
```

Both moving are not changed in the **faster model** except for using the position of actors stored in the global arrays when needed.

2.5 Templates

2.5.1 Initializer

The *Initializer* template is a simple automaton with three states and two edges used to set up the map and then start all other agents.

```
void init_map() {
  // Fires
  map[4][3] = CELL_FIRE;
  map[4][4] = CELL_FIRE;
  ...

  // Exits
  map[0][4] = CELL_EXIT;
  map[0][5] = CELL_EXIT;
  ...
}
```

Its initial state is a committed state, meaning that the model must follow one of its edges right away. Its only edge runs the function `init_map()`, which configures the position of fires and exits in the map 2D array representation (agents will later set their position on their own). One could initialize the array in place in the *declarations*, but with increasing map sizes, it would become unmanageable. The use of a function allows for a clear definition of where entities are placed.

Then the *Initializer* has a second committed state with one edge that triggers the `init_done` broadcast channel. All other agents have their initial state with a single arc that synchronizes them on the `init_done` channel. When the channel fires, all the agents perform a simple initialization step (e.g., they set their position in the map) and then they become ready for the simulation to properly start.

2.5.2 Survivor

At the beginning of the simulation, *survivors* position themselves on predetermined coordinates in the map. If they are near a fire, they become survivors *in-need* of assistance; otherwise, they are considered normal *survivors*.

Survivors *in-need* cannot move, and if they are not assisted within a certain time period T_v, they become casualties. However, if they receive assistance within the time period, they become safe. This behavior is modeled by setting bounds on the *survivor*'s clock. When the time period T_v is exceeded, the model transitions to the `Dead` state. In both cases, the *survivors* leave the simulation, freeing the map cell they were occupying.

Other *survivors* who are not near a fire default to moving towards an exit, following their designated *moving policy*. This movement can stop in four cases:
- If they move within a one-cell range of an exit, they become safe and leave the simulation, freeing the map cell they were occupying.
- If they receive an instruction from a *drone* to directly assist someone *in-need* or call a *first-responder*, they stop targeting an exit and start following the instruction. In both cases, the *survivor* reaching the new target is modeled by waiting for a duration equal to the distance to the target, rather than actually moving on the map. Although this does not accurately model the simulated scenario, particularly the interaction between moving agents on the map, it is necessary to keep the model simple and maintain acceptable verification times.
- When they have no available moves. This could be due to the map topology blocking the *survivor*'s path or the moving policy not allowing any moves. For example, the `DIRECT` moving policies presented earlier can potentially lead to a

survivor being stuck in a loop where moving around an obstacle frees the previous cell, which is then re-selected. We deemed these cases acceptable because we considered it reasonable for the map topology to present challenges and for civilians to struggle in finding the proper path.

Note that we built the model such that *survivors* will never move near a fire, thus they cannot become *in-need* during the simulation.

2.5.3 First-responder

First-responders defaults to moving towards the nearest survivor *in-need*, but can stop moving in 2 cases:

- When they reach the targeted survivor *in-need* they start assisting. After T_{fr} the assistance is completed and the survivor *in-need* is considered safe;
- When they are asked by a *survivor* to assist someone *in-need*, they stop moving to wait for the *survivor* to reach them. This is modeled with a wait equal to the distance between the *survivor* and the *first-responder*.

2.5.4 Drone

Drones are equipped with vision sensors capable of detecting *survivors* within a predetermined range N_v. When they detect both a "free" *survivor* (a.k.a. *zero-responder*) and someone *in-need*, they can instruct the *survivor* to assist.

When a possible *zero-responder* and someone *in-need* are in range of the sensors, the *drone* start a sequence to select the agents to involve in a particular command. It first selects the possible *zero-responder*, it selects the survivor *in-need* and then, depending on whether there is at least one *first-responder* available or not, decides whether to make the *zero-responder* assist directly or making him call a *first-responder* which is then selected.

Drones also have a fixed moving pattern that follows a predetermined path. This path is decided prior to the simulation and is not influenced by the state of the map or the agents. This is a simplification to keep the model complexity low and to avoid the need for the *drone* to plan its path dynamically. The current path is a square with a parametric side length, each *drone* can be setup with a different dimension and with a specific starting position.
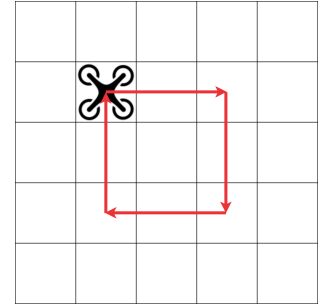
Drone moving pattern

In the **faster model** after selecting the actor needed to perform an action the *drone* sends a message to all the actors involved with the total waiting time needed to complete the action.

2.6 Statistical Model Checking

All the optional stochastic features have been implemented:

- The *survivors* acknowledge the instruction and enact with probability S_{listen}, and ignore it (or miss it) with probability $1 - S_{\text{listen}}$. We assume *survivors* share the same behavior, hence the same probability is used for all of them.
- The *drones* vision sensors fail with probability P_{fail}. We assume *drones* share the same sensors, hence the same failure probability is used for all of them.

3 Scenarios & Properties

To highlight the strength and weaknesses of the model, we have defined a set of scenarios that we have simulated trough it. The scenarios are designed to test the model in different conditions, such as the presence of multiple fires, the distribution of agents, and the effectiveness of the moving policies.

For all models, we have established the following parameters, which are scenario-independent and cannot be controlled:

- $T_v = 30$: The time before an *in-need* becomes a casualty;
- $T_{\text{zr}} = 10$: The time a zero-responder needs to assist an *in-need*;

The other parameters are changed to illustrate the functionality and efficiency of the system.
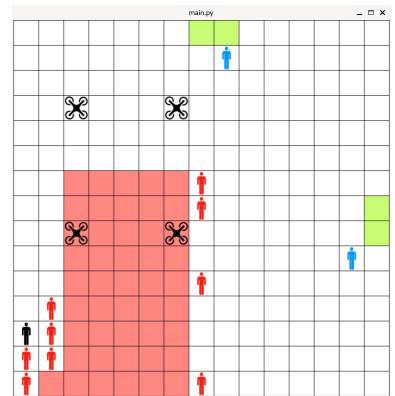
For each scenario, we calculated:

- $N\%_{\text{max}}$: The maximum percentage of safe individuals over the total number of survivors (checked by: "sup{safe_survivors + dead_survivors == N_SURVIVORS}: safe_survivors")
- $N\%$: The guaranteed number of safe individuals (checked by "inf{safe_survivors + dead_survivors == N_SURVIVORS}: safe_survivors")
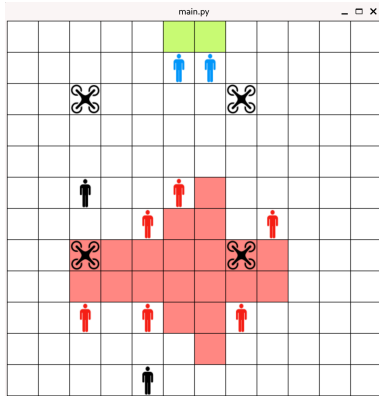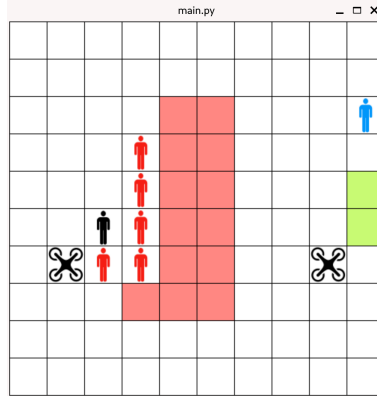
| Plane crash | Lone survivor | Dividing branches |

3.1 Plane Crash

Plane goes kaboom.

3.2 Lone survivor

During a fire a *first-responder* is called to save as many lives as possible. Due to the geometry and the poor ventilation the space is full of smoke, impeding the *first-responder* ability to see any in need directly (moving policy random). In this scenario,

without drone assistance, the *first-responder* is able to save at most $N\%_{\text{max}} = 83.33\%$ and at minimum $N\% = 16.67\%$ depending on the random path.

By activating the system with $N_v = 1$ and $N_r = 2$, the *drone* instruct a survivor to bring a *first-responder* to the in-need. Effectively the *first-responder* needs to wait more time, but it always reach the group of in needs obtaining $N\%_{\text{max}} = 66.67\%$ (lower due to wait) but improving the minimum saved to $N\% = 50\%$.

In this scenario, the *first-responder* is too slow to reach and heal the in-need to save more individuals. Since we cannot change the speed of the *first-responder*, we can only improve the assist time to $T_{\text{fr}} = 1$, (in a real-world scenario, this could be achieved through better training or better equipment), resulting in $N\%_{\text{max}} = 100\%$ and $N\% = 83.33\%$. This is the configuration presented in faster_model_scenario_2.

This scenario is designed to test the effectiveness of the model in bringing a *first-responder* (moving randomly) to a group of survivors *in-need*. A single *survivor* is placed in a corner of the map near a group of *in-need*, with the *first-responder* located in the opposite corner. The *survivor* is tasked by the *drone* to fetch the *first-responder* and must navigate through the map to reach them and assist the *in-need* individuals. After that, the *first-responder* will start helping all the *in-need* survivors nearby until either all are dead or safe.

3.3 Divided branches

First responders arriving on the scene finds a wall of fire dividing the space in two. Due to their training, all the *first-responders* moves to the nearest *in-need* (Moving policy `DIRECT`), even when seeing another *first-responder* going in that direction. This causes the *first-responders* to focus first on the nearest group of survivors, and then on the group on the other side of the wall, reaching it too late.

This scenario is designed to test the effectiveness of the model in bringing *first-responders* to all groups of *in-need* individuals, increasing the chance of more survivors reaching safety. When using the policy `DIRECT`, *first-responders* try to reach the nearest *in-need*, and after assisting him they go to the next closest, and so on. Without *drones* assistance, all the *first-responders* would go to the same group of *in-need*, leaving the others to die. With our system, the *first-responders* get split assisting both groups.

As before, we first check the survivor rate of the model without drones, obtaining $N\%_{\text{max}} = 60\%$ and $N\% = 40\%$.

By turning on the system with $N_v = 1$ and $N_r = 2$, we obtain $N\%_{\text{max}} = 60\%$ and $N\% = 50\%$ improving the result, but not by much. This highlights a weakness of the system: the drones always prefer the *first-responder* when available, even if they are very far away, keeping them occupied longer. In this case to further improve the survival rate either more *fist-responders* are needed or a better decision policy for the *drones* has to be implemented.

4 Conclusion

In this project, we developed a model using Uppaal to simulate search-and-rescue operations involving survivors, first responders, and drones. The model is quiet flexible and almost every parameter can be changed to suit most simulations.

Developing two models allowed us at first to create a complex but true to life model, capable of simulating easier scenarios in a more expressive and easy to understand way, and then by simplifying it to understand what was essential and what could be removed in order to speed up simulation time while maintaining the same core functionalities.

Statistical model checking introduces uncertainty allowing us to understand the model's behavior in a more realistic scenario, where agents can make mistakes or fail to perform their tasks.

In conclusion, our graphical interface allows us to easily create different scenarios and our model enable us to study them, maximizing the survival rate by assessing the effect that changes in different parameters have on the system.

5 Appendix

5.1 Simulation Graphical Tool

To facilitate the development of the model we've created a graphical tool capable of visually represent and interact with the simulation model. The tool supports 3 working modes: editor, trace visualizer and live visualizer.

Creating a simulation scenario requires deciding on various parameters, such as the map dimensions, agent placements, locations of fires and exits, and specific agent parameters like a *drone*'s vision range. To facilitate the placement of entities on the map, the graphical tool can be launched in editor mode. This mode enables selective placement of entities through a combination of mouse and keyboard inputs:

- Clicking the mouse places an entity in a cell.
- Clicking again cycles through different available entities, with the cycling direction determined by the use of either the left or right mouse button.
- Clearing a cell can be accomplished by clicking the middle mouse button.
- *Drones* can be placed by holding the shift key while clicking.
- To draw over a larger area, move the mouse while keeping the button pressed.
- Once the map design is complete, pressing `CTRL+S` generates the code required for integration into the Uppaal model.
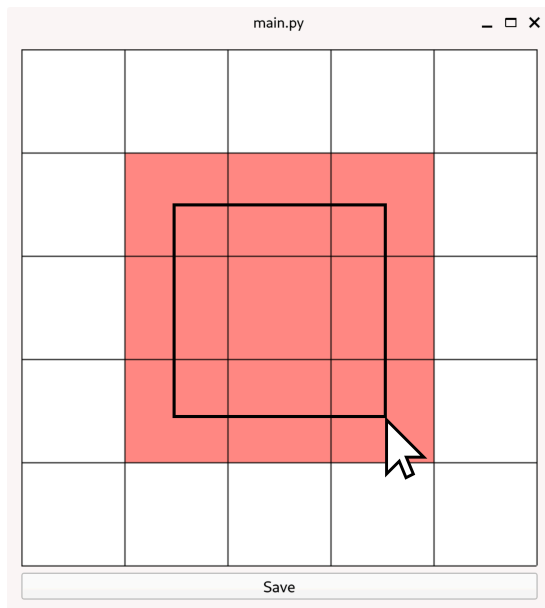
Editor drag operation
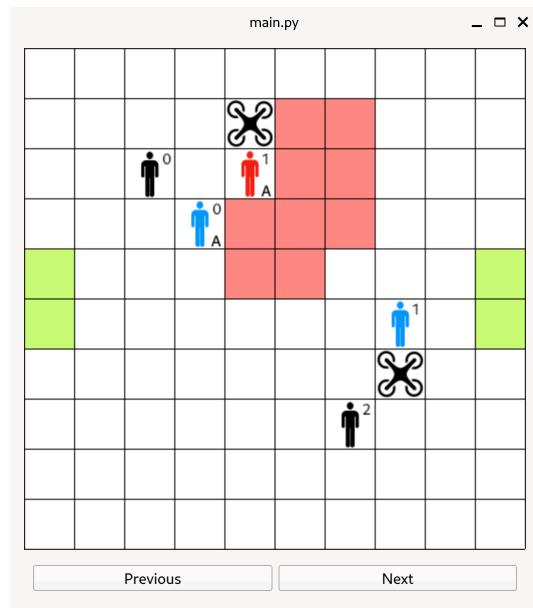
Trace visualizer

When performing simulations in Uppaal, understanding how the scenario evolves can be really challenging. This is because through the Uppaal's interface you have access to global and template's local variables, which can be hard to decipher at a glance. To ease the understanding of the simulation, the graphical tool is able to visualize a trace file saved through the *Symbolic Simulator*. The interface visualizes for each simulation step the complete status of the map, indicating the agents' index with a number and their status (e.g. when *first-responders* are busy assisting, an "A"

appears). The user can move through the simulation with the "Previous" and "Next" buttons or with the arrow keys.

Since the live visualizer proved to be very useful, we decided to further extend its capabilities by supporting the live visualization of the simulation. To achieve this functionality, we needed a way to make an external application talk to Uppaal, which can be accomplished thanks to Uppaal's external functions. This feature allows Uppaal to call a function coded in another language during the simulation, and is implemented by dynamically linking a user-provided library. Our tool provides a simple function (`send_state_via_post_request(...)`) that sends the map status via a `POST` request to a local endpoint. This function is called at each model's update thanks to the `before_update` and `after_update` statements. The endpoint is a simple web server that runs alongside the graphical tool, that listens for the requests and updates the visualization accordingly. The tool's live visualization feature allows seeing both the symbolic or concrete simulation in real-time!