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**

Advisor: Prof. Pierluigi San Pietro
Co-advisors: Dr. Livia Lestingi
Academic Year: 2023-24

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 amount 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 simulation.

The key characteristics of the agents are the following:

- **Survivors**: They can be in 3 different states, depending on whether they find themselves near a fire or if they are following instructions.
  - **In-need** (i.e. near a fire): They cannot move and need to be assisted. After T_v time units, they became a casualty and die.
  - **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 instructions, 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 their sensors, 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).
- Diagonal movements from one cell to another are allowed. Therefore, the distance can be easily computed as the maximum between the difference of the x and y coordinates.
- The movement of *survivors* and *first-responders* towards a human target (e.g. a *survivor* goes to a *first-responder*) is modeled with a wait state, where the agents remain idle for the duration of the movement, and then the coordinates are set to the target position. 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 would play out in real life, we only model the necessary parts of it. Instead of the actors moving on the map, we model the movement as a wait states. This means that when an agent is instructed to move to a certain position, it will wait for a specific amount of time before "teleporting" to the target position. If a movement is composed of different steps (i.e. *zero-responder* goes to call a *fist-responder* and then back to the *in-need*), it is now modeled as a single wait state that waits for the time needed to travel the full path. 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 computations, frequently used values are stored in global variables for quicker access during iteration. For instance, instead of scanning the whole map grip for the position of an actor, we store positions in a global array indexed by the survivor ID and access the array sequentially, turning a $O(n^2)$ operation into a $O(n)$ one.

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, 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, losing the ability of specifying different parameters for each agent, but reducing the complexity of the drone template.

2.2.1 Map Representation

Despite each agent holding its own position internally, a global representation of the map is needed for agents that 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 the type of human agent occupying it. This avoids each agent needing to hold 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 previously occupying and sets the new cell to 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 than in the map, the position of each actor (and exit) is stored in an array and updated at each movement to reduce calculations in the following templates.

2.3 Synchronization and Message Passing

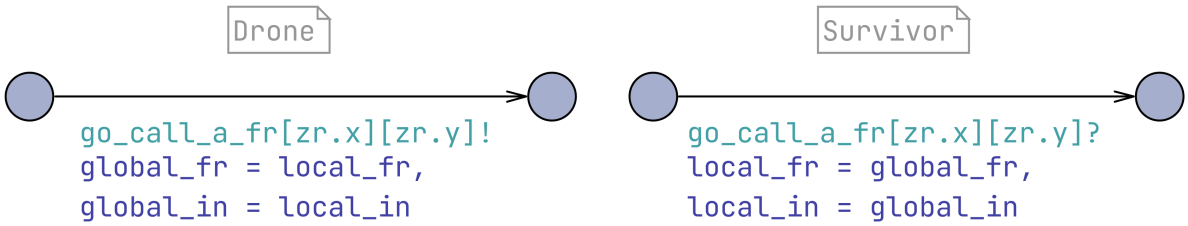
Between the different agents, there are some interactions that require a way to pass a payload. For example, when a *survivor* is instructed by a *drone* to contact a *first-responder*, the *survivor* must receive from the *drone* both the positions of the *first-responder* and that of the one *in-need*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



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

All the moving policies work 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.

A move is considered valid if the target is inside the map bounds and the cell is empty, i.e. not occupied by a fire or another agent.

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, for the selected moving policy.

To try out 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 only enables the moves with the lowest direct distance to the target. As shown by Figure 3, if more than one cell is at the lowest distance, all of them are enabled and a random one among those will be select.

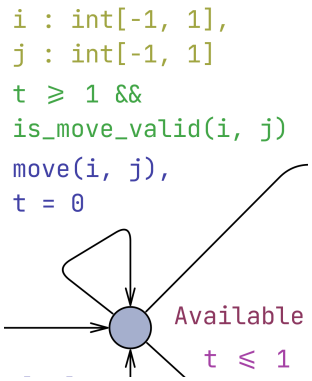


Figure 1: Moving edge

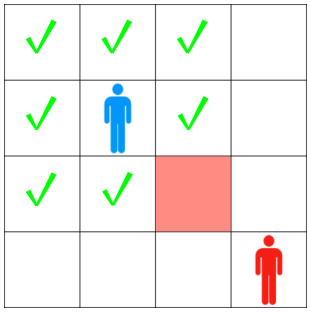


Figure 2: Moving policy `RANDOM`

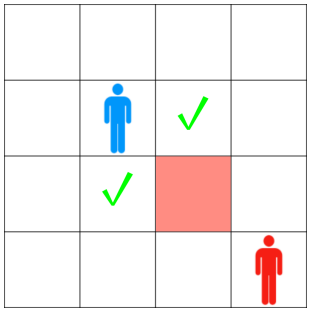


Figure 3: Moving policy `DIRECT`

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 policies remain unchanged in the **faster model**, the only difference being using the position of actors stored in the global arrays for quicker access.

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 so that *survivors* will never move near a fire, therefore they cannot become *in-need* during the simulation if they do not start as such.

2.5.3 First-responder

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

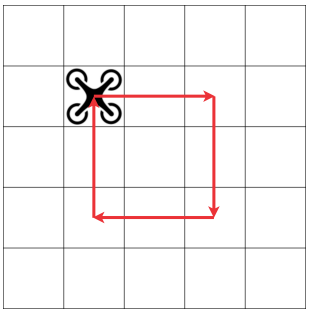
- When they reach the targeted survivor *in-need* they start assisting. After T_{fr} the assistance is completed and the survivor *in-need* is considered safe.
- When they are asked by a *survivor* to assist someone *in-need*, they stop moving to wait for the *survivor* to reach them. This is modeled 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 pre-determined range N_v. When they detect both a "free" *survivor* and one *in-need*, they instruct the *survivor* to assist.

To instruct a *survivor*, the *drone* enters a sequence of states to select the agents to involve in a particular command. It first selects the "free" *survivor* and the survivor *in-need*, then depending on the number of available *first-responder*, decides whether to make the *survivor* act as a *zero-responder* and assist the *in-need* directly, or select a *first-responder* to be called by the *survivor*, which will then perform the assistance.

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 all the actors to instruct, the *drone* calculates the total wait time for each actor involved and sends it to them.

3 Scenarios & Properties

To highlight the strength and weaknesses of the model, we have defined a set of scenarios that are designed to test it under 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*.

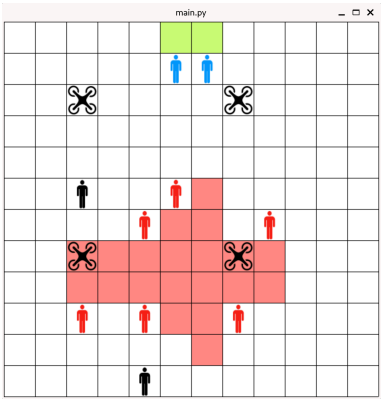
The other parameters are tuned to illustrate the functionality and efficiency of the system.

We always kept T_{scs} at 60 time intervals to allow the system to reach a stable state before the simulation ends. For each scenario, we calculated:

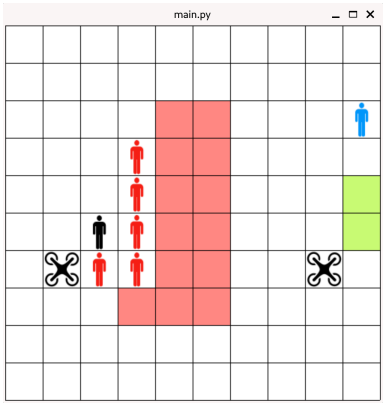
- $N\%_{\text{max}}$: The maximum percentage of safe individuals over the total number of survivors within T_{scs}.
- $N\%$: The guaranteed number of safe individuals within T_{scs}.

All the optional stochastic features have been implemented:

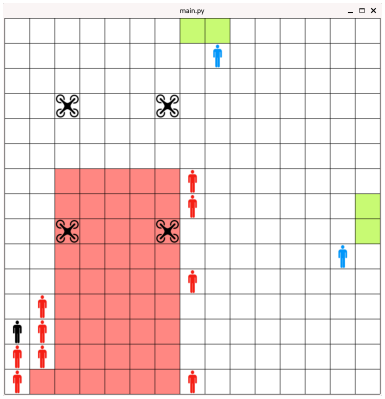
- The *survivors* acknowledge the instruction and enact with probability S_{listen}, and ignore it (or miss it) with probability $1 - S_{\text{listen}}$. We assume *survivors* share the same behavior, hence the same probability is used for all of them.
- The *drones* vision sensors fail with probability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



Plane crash



Lone survivor



Dividing branches

3.1 Plane Crash

A plane crashed and it is currently on fire. Passengers exited the plane and are scattered around the map. A single ambulance arrives on the scene, providing the *survivors* with one exit spot. The area is free of obstacles and *survivors* and *first-responders* can clearly see their surroundings, therefore they are configured with the policy `DIRECT`. The scenario is considered to vary depending on two factors:

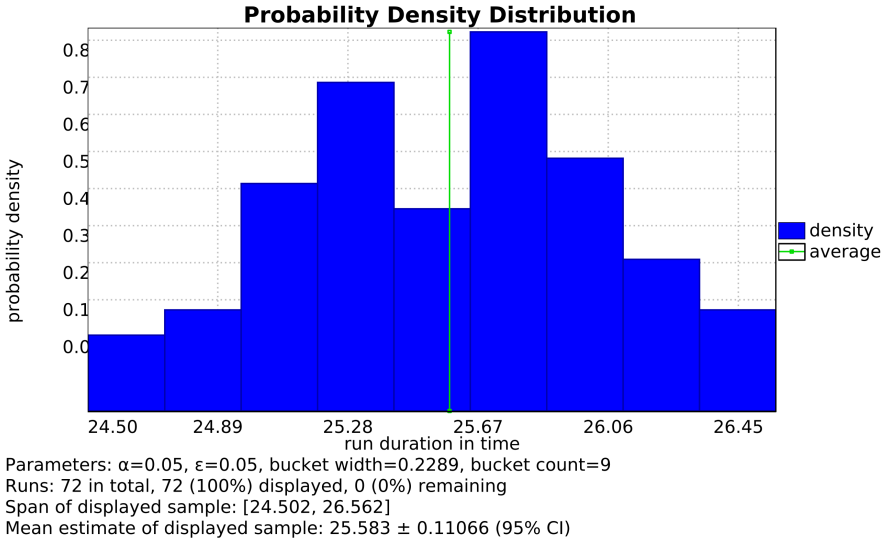
- The *drones* vision range depends on the environment. If the incident is in an open field, the *drones* have a larger vision range, while in a forest, the vision range is smaller.
- The ambulance staff may not be prepared to directly assist the *survivors*, therefore *first-responders* may not be available.

$N_{\text{SURV.}}$	N_{FR}	N_{DRONES}	N_v	T_{fr}	T_{zr}	T_v	**min** $N_{\%}$	**max** $N_{\%}$
8	0	0	-	5	8	30	25%	25%
8	0	4	1	5	8	30	25%	25%
8	0	4	2	5	8	30	37,5%	50%
8	2	0	-	5	8	30	100%	100%
8	2	4	1	5	8	30	100%	100%

Table 1: Plane Crash results

The plane crash scenario emphasizes the importance of *first-responders* in ensuring the safety of survivors. Without *first-responders*, when more civilians are *in-need* rather than not, there will always be someone *in-need* that cannot be brought to safety. In this case the presence of *drones* allows to save some lives, but this is not enough to ensure the safety of all the survivors. When instead *first-responders* are present, all the survivors can be saved and drones could be superfluous depending on the number of *first-responders*, their training level and moving policy.

Since *drones* are superfluous in this scenario, *first-responders* are not influenced by the failure of the drones sensors nor the probability of *survivors* listening to instructions, therefore running the SMC model yields $N\%_{\text{max}} = N\% = 100\%$.



Probability of all survivors being safe after T_{scs}

3.2 Lone Survivor

During a fire, one *first-responder* is called to save as many lives as possible. Due to the particular topography and the lack of wind, the space is full of smoke, impeding the *first-responder* ability to see anyone directly (moving policy `RANDOM`). On the contrary, the *survivors* are locals and can navigate the space even with their eyes closed (moving policy `DIRECT`).

$N_{\text{SURV.}}$	N_{FR}	N_{DRONES}	N_v	T_{fr}	T_{zr}	T_v	**min** $N_{\%}$	**max** $N_{\%}$
6	1	0	-	5	8	30	16.7%	83.3%
6	1	2	1	5	8	30	50%	66.7%
6	1	2	2	4	8	30	66.6%	83.3%
6	1	2	2	3	8	30	66.6%	83.3%
6	1	2	2	2	8	30	83.3%	100%
6	1	2	2	1	8	30	83.3%	100%

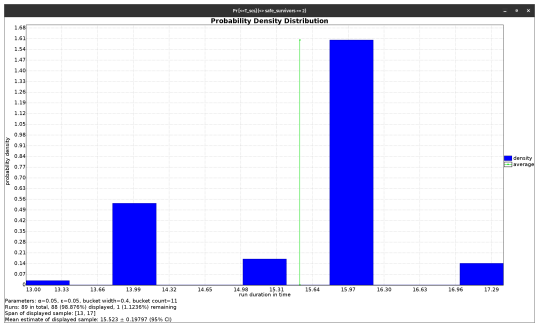
Table 2: Lone Survivor results

In this scenario, without *drone* assistance, the *first-responder* is able to save only a limited number of lives. This depends heavily on his inability to clearly see the survivors and reaching them directly. In very rare cases, the *first-responder* is lucky enough to reach the *survivors* in a very short set of moves. This is reflected by extreme values of $N_{\%}$.

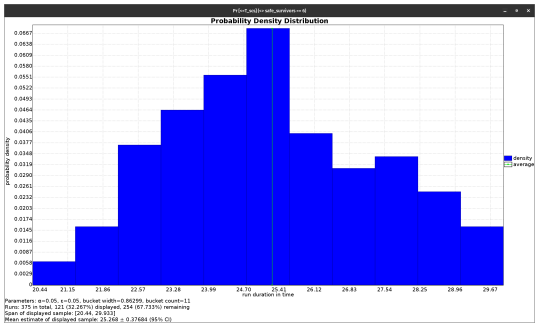
By deploying two *drones*, *survivors* are instructed to reach out to the *first-responder*, and bring him close to the *in-needs*. In this case, the *first-responder* will reach the group of *in-need* consistently more often. This results in an higher $N_{\%\text{min}}$, meaning that it is guaranteed that more *survivors* will always be saved, but a lower overall survival rate is achieved, which gives a lower $N_{\%\text{max}}$. We can attribute this to the fact that the *first-responder* "wastes" time to wait for the survivor to reach him, when he could be moving towards the *in-need* directly, had he been using the policy `DIRECT`.

Due to a single *first-responder* being present on the scene, a lower assist time T_{fr} is crucial to save more individuals: lowering T_{fr} increases the survival rate, as shown in Table 2.

Setting the probability of the *drones* sensor to fail to 10% and the probability of the *survivors* to listen to the instructions to 40%, we can guarantee that, within T_{scs}, at least 2 survivors will be safe ($N\% = 33\%$) with a probability of 96.93%. The maximum possible number of survivors saved is 6 with a probability of $N\%\,\text{max} = 32.40\%$.



2 safe



6 safe

3.3 Divided Branches

First-responders arriving on the scene find a wall of fire that splits the area in half. Multiple groups of *in-needs* are spread on the two sides of the fire. Due to their

training (modeled by applying policy `DIRECT`), all *first-responders* move to the nearest *in-need*. Survivors are also assumed to use the moving policy `DIRECT`.

Description	$N_{\text{SURV.}}$	N_{FR}	N_{DRONES}	N_v	T_{fr}	T_{zr}	T_v	**min** $N_{\%}$	**max** $N_{\%}$
No drones	10	2	0	-	5	8	30	40%	60%
1 drone	10	2	2	1	5	8	30	50%	60%

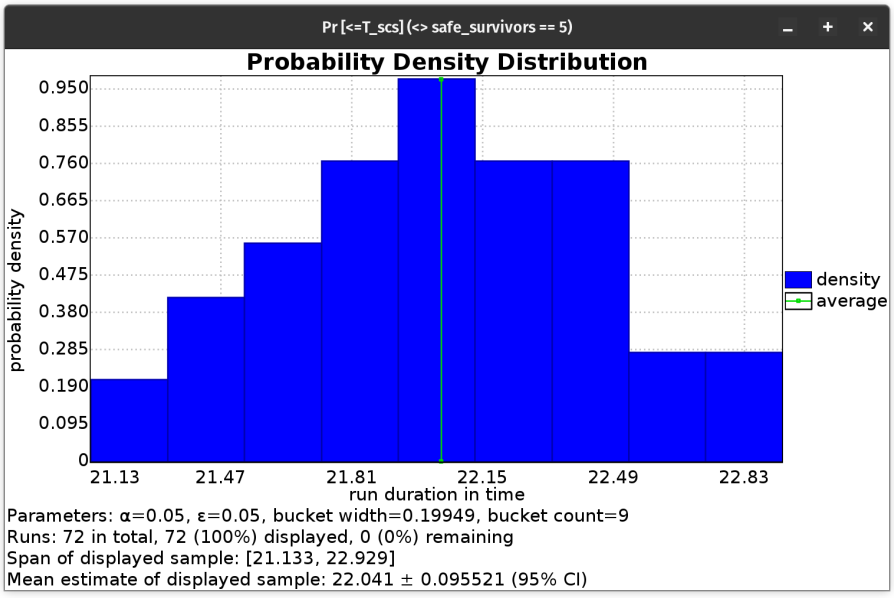
Table 3: Divided Branches results

This scenario is designed to test the effectiveness of the model in bringing *first-responders* to all groups of *in-needs*. With the moving policy `DIRECT`, *first-responders* try to reach the nearest group, and move around until all the *in-need* individuals of that group are brought to safety. Even if enough *first-responders* are deployed to save all *survivors*, they will be stuck on the first group while trying to assist them, and cannot reach the other groups within T_{scs}.

When drones are deployed, *survivors* that are not in need of assistance are instructed to reach the *first-responders* to bring them to their group. This effectively spreads out the *first-responders*, solving the policy `DIRECT`'s limitation. This is evidenced by an higher $N_{\%\text{min}}$, meaning that more *in-need* are guaranteed to be saved.

Although an improvement is achieved, the overall survival rates are not improved 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.

For the reasons we just described, drones don't seem to be affecting stochastic queries for this scenario, as we obtain $N\%_{\text{max}} = N\% = 50\%$ in both cases.



Probability of all survivors being safe after T_{scs}

4 Conclusion

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

Developing two models allowed us to first create a complex but true-to-life system, capable of simulating simpler scenarios in a natural way that was easy to follow. We then proceeded to simplify it to keep the essential parts only, to lower simulation cost and verification time dramatically while maintaining functionality.

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

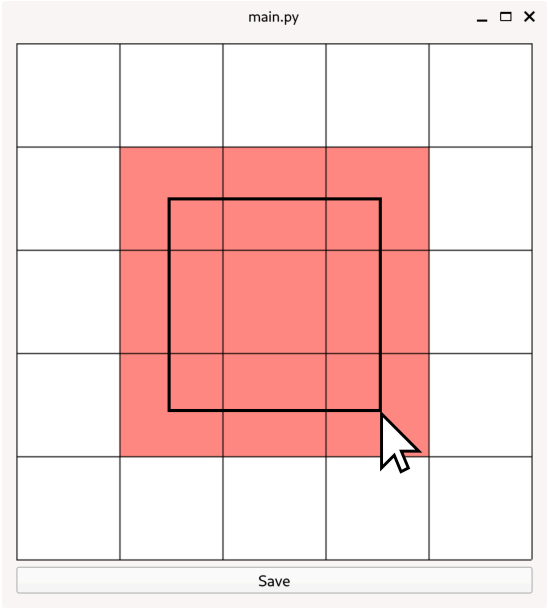
5 Appendix

5.1 Simulation Graphical Tool

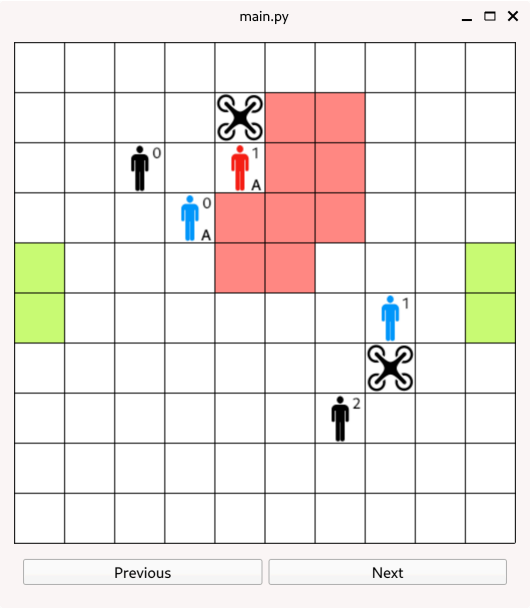
To facilitate the development of the model we've created a graphical tool capable of visually representing and interacting with the simulation model. The tool supports 3 working modes: editor, trace visualizer and live visualizer.

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!