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* surveys 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 placed 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 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 it's own parameters (e.g. `N_v[id]`).

This setup allows to easily define the simulation parameters all inside the *Declarations* section, thus without modifying neither the templates nor the *System declarations*, and to easily assign different parameters to each template instance.

2.1.1 Map Representation

Desipite 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);
}
```

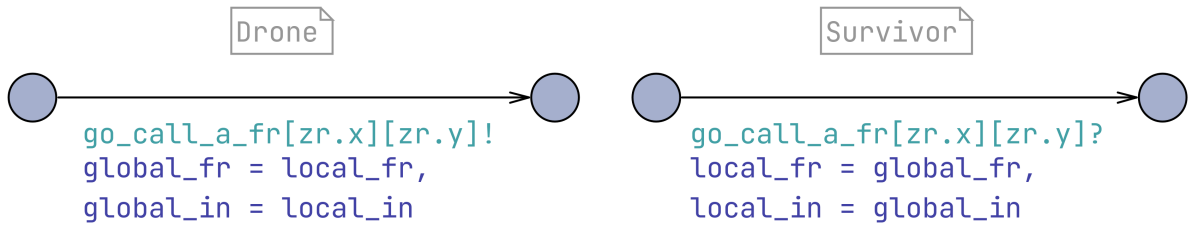
2.2 Syncronization and Message Passing

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 at a channel on 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*.



2.3 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 offets `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 impleted 2 simple policies:

- The **random** policy simply checks if the move is feasible (i.e. wheter 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 have the same distance to the target, the policy enables all of them and the model will randomly select among those.

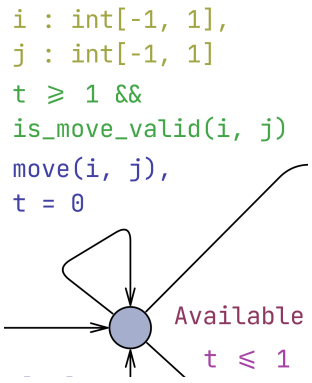


Figure 1: Moving edge

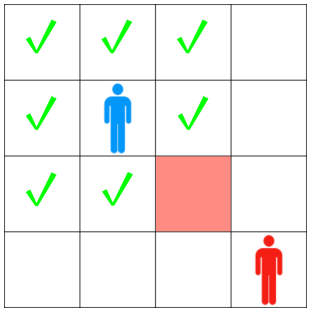


Figure 2: Moving policy `RANDOM`

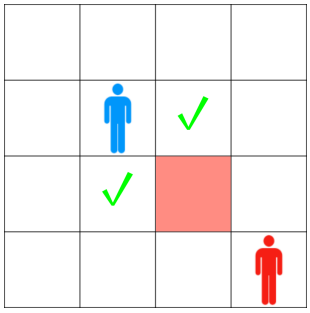


Figure 3: Moving policy `DIRECT`

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;
}
```

2.4 Templates

2.4.1 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.

2.4.2 Surivor

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 in the map. Although this does not accurately model the simulated scenario, particularly the interaction between moving agents in 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 reselected. 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.

2.4.3 First-responder

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

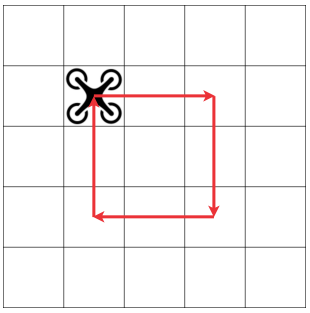
- When they reach the targeted survivor *in-need* they start assisting. After T_{fr} the assistance is completed and the *in-need* survivor is considered safe;
- When they are asked by a *survivor* to assist someone *in-need*, they stop moving to wait for the *survivor* to reach them. This is modeled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

2.4.4 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-risponder* 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.

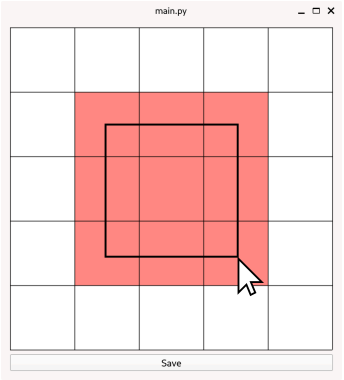


Drone moving pattern

3 Simulation Graphical tool

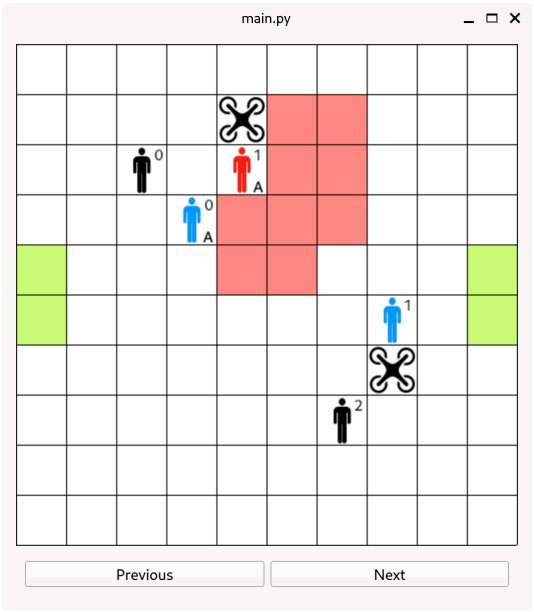
To facilitate the development of the model we've created a graphical tool capable of visually represent and interact with the simulation model. The tool supports 3 working modes: editor, trace visualizer and live visualizer.

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:



Editor drag operation

- 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.



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 decifer 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, idicating 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 proven to be very usefull, 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 Uppall, which can be accomplished thanks 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 to see both the symbolic or concrete simulation in real time!

4 Properties

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut.

5 Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut.