Summary

This paper addresses the challenge of enabling drones to autonomously navigate and map unknown environments, specifically targeting the identification and localization of pumps within a room. This research aims to enhance the efficiency of current manual scanning methods performed by technicians. The study is particularly relevant to industrial applications, such as those needed by Grundfos, a global pump manufacturer. Grundfos has therefore directly contributed to this study, by presenting the case study and making a pump-detection neutral network available.

The system employs UPPAAL STRATEGO to model the problem as a Markov Decision Process (MDP). UPPAAL STRATEGO applies Q-learning to derive a near-optimal policy for the drone. Safety mechanisms, including a learning and runtime shield, are implemented to prevent collisions. The drone is equipped with LiDAR and IMU sensors, which provide odometry data crucial for simultaneous localization and mapping (SLAM) through the Robot Operating System (ROS) and the Slam Toolbox framework.

The Stochastic Model Predictive Controller (STOMPC) framework is used to handle uncertainties in the environment by updating the MDP in UPPAAL STRATEGO with real-time state information. The proposed approach is validated using the Gazebo Simulator. The RL-based method requires fewer actions to complete mapping tasks but shows slightly worse time efficiency than a baseline algorithm using Breadth-First-Search. This is due to the bottleneck of training. The proposed method is also shown to work in multiple environments of different sizes and shapes. Tests on a real-world TurtleBot3 robot confirm the method's robustness and applicability across different robotic platforms.

To further enhance the system, future work suggests looking into how the map of the environment represented by a matrix in UPPAAL STRATEGO can be optimized.

In conclusion, this study highlights the feasibility and effectiveness of using reinforcement learning in collaboration with STOMPC, ROS and Slam Toolbox to automate drone inspections. The findings found in this paper can contribute to more efficient and reliable automated inspection systems in the future.

Exploring Unknown Environments with UPPAAL STRATEGO: Reinforcement Learning for Drone Navigation and Pump Localization

Thomas Grubbe Sandborg Lauritsen
Department of Computer Science
Aalborg University
Aalborg, Denmark
tlauri19@student.aau.dk

Magnus Kallestrup Axelsen
Department of Computer Science
Aalborg University
Aalborg, Denmark
maxels19@student.aau.dk

***Abstract*—We consider the problem of using autonomous Unmanned Aerial Vehicles (UAVs), also known as drones, to explore, map and find points of interest (POIs) in an unknown room. The paper proposes employing Reinforcement Learning (RL) to enable the drone to map and explore these unknown rooms. We propose modelling the problem as a Markov Decision Process (MDP) in UPPAAL STRATEGO, which utilizes Q-learning to synthesize a near-optimal policy. This policy will be used to generate a sequence of actions that will be activated on the drone. Additionally, we implement the framework STOMPC as a stochastic model predictive controller, to capture the uncertainties of the room and the dynamics of the drone. STOMPC achieves this by giving UPPAAL STRATEGO updated information about the new true state of the drone after activating all the actions in the sequence. We also employ two different shields, a learning shield and a runtime shield, used to enforce safety constraints on the actions the drone can take. The drone used in this work is equipped with a LiDAR sensor and an IMU sensor providing odometry data. We employ Robot Operating System (ROS) to control the drone. ROS also provides us with a simultaneous localization and mapping (SLAM) framework called Slam Toolbox, which we use to update the map given to UPPAAL STRATEGO. To validate our proposed approach, we use the tool Gazebo Simulator to simulate an X500 drone and the room that the drone should map and explore. We compare our approach to a Breadth First Search (BFS) based approach. We show that our approach manages to fully explore a room and examine all POIs with 33 fewer actions activated on average while using marginally more time. Options to further reduce the completion time for our approach are also presented. We also show the generality of our approach, by mapping and exploring rooms of different sizes and shapes. Lastly, we show the proposed method working on a real-life TurtleBot3 robot.**

Fig. 1: Pump wall with Grundfos products

1. INTRODUCTION

In recent years drones, also called Unmanned Aerial Vehicles (UAVs), have emerged as a new tool, to automate various tasks in different industries. This can be seen anywhere from agriculture where drones can be used for crop management [1], to construction where they can be used for safety management [2] among many things. Grundfos, a global leader in water solutions, has begun looking at how UAVs can be used to automate their tasks. Grundfos has pump installations all around the world, that require check-ups and maintenance, involving a technician visiting the site to perform inspections. The technician will use a mobile application to perform a scan of the installation. The application uses augmented reality, allowing the technician to place markers that indicate specific points of interest such as pumps. When finding a pump, or any other point of interest, the technician must make sure to capture it from a sufficient amount of angles, to increase the quality of it. For instance, each pump will have a nameplate located on it, enabling the technician to identify the pump. When the scanning has been completed, the mobile application will output a 3D map of the installation. The pumps in the 3D map can be clicked, to show additional information such as their nameplate. Points of interest such as pumps, can be seen in fig. 1.

In this paper, we present a case study in collaboration with Grundfos, where we aim to leverage the use of UAVs to map and localize pumps in a previously unknown environment. Rather than having a sales representative or technician sent to the site to perform this scan, it should be done by a UAV. Ideally, Grundfos can send a drone to the installation site and ask the customer to place the drone in the environment and turn it on. The drone will then take off, and perform the inspection automatically without any need for human intervention. The drone must therefore be able to locate all the pumps in the installation, map the installation, and be able to do it safely, meaning that the drone must not crash into anything. Meanwhile, the drone will not have any prior knowledge about the installation, including the size of it and how many pumps there are in it.

To solve this problem, we propose using a reinforcement learning approach to allow the drone to learn how to navigate and locate pumps in an unknown environment. This will be done by formulating the drone's behaviour as a Markov decision process in UPPAAL STRATEGO [3], and using UPPAAL STRATEGO to compute a strategy using reinforcement learning. We will be using Robot Operating System (ROS) [4] to control the drone and execute the commands from the strategy. To map the environment a SLAM-framework called Slam Toolbox [5] will be integrated into the platform, allowing for real-time mapping of the environment. We will use the Gazebo Simulator [6] for drone and environment simulation [7] and show the proposed method working on a real-world Turtlebot3 [8] robot as proof of concept. All of this will be connected using a framework called STOMPC [9]. The idea is that STOMPC will send the current position of the drone and the current knowledge of the map, as an initial state to UPPAAL STRATEGO. UPPAAL STRATEGO will then compute a strategy based on this initial state, and send it back to STOMPC. The case study presented by Grundfos will from now on be denoted as the Grundfos Problem (GP).

2. Related Works

The task of exploring an unknown environment while finding specific areas of interest, has been researched extensively in the field of robotics. In recent years, researchers have introduced learning-based exploration algorithms, dividing exploration algorithms into two groups: traditional exploration and learning-based exploration. A particular well-known traditional exploration algorithm is the one presented by Yamauchi [10]. The paper proposes a frontier-based approach to exploring an unknown environment. The basic idea of the frontier-based approach is to place so-called frontiers, at the boundaries of known and unknown space. The autonomous vehicle will then move to the nearest frontier, expanding the map. The algorithm still performs well when compared to newer learning-based algorithms [11], but it can not always guarantee safety, since the locations of the frontiers may be too close to an obstacle.

Lately, researchers have attempted to use reinforcement learning (RL) to tackle the task of exploration [12], [13]. In RL-based exploration, the algorithm will use the current state of the environment and sensor data to reason about how much information is gained by each action. While learning-based methods are highly adaptive, they can suffer from being inefficient due to long learning times. Researchers have tried to combat this limitation with exploration, by having a deep reinforcement learning-based algorithm learn an exploration strategy on a partial map. This means that for exploration, it is only needed to train once and thereafter simply use the neural network to calculate Q-values from [11]. However, the experiments from this work are all performed on maps very similar to the partial map used for training. This may reduce the learned strategy's quality in more generic environments.

Work has also gone into utilizing the tool UPPAAL STRATEGO, which uses Q-learning to synthesize plans under learned policies for a fleet of robots. The researchers explore using UPPAAL STRATEGO for dynamic route planning with Autonomous Mobile Robots (AMRs), where a fleet of AMRs perform tasks in a shared and known environment [14]. UPPAAL STRATEGO was used to synthesize an individual plan for each AMR about where a task is picked up and where it should be delivered in the environment, respecting where all the other AMRs are. When the AMRs are performing their respective plans, the AMRs will receive updated information about the environment and synthesize a new plan based on this. The research shows promising results using UPPAAL STRATEGO for model predictive control and planning over a greedy approach. However, in this work, AMRs are moving along edges between certain points in the environment, which needs to be known beforehand, whereas, in our work, both the environment and where the drone can move is unknown. UPPAAL STRATEGO have also been used for stochastic model predictive control in areas such as controlling traffic lights in an intersection, where stochasticity comes from the number of cars that approach the intersection. It has also been used for controlling floor heating in a house where windows can open and close randomly as well as other factors contributing to the stochasticity in the environment [9], [15]. These problems are comparable to the problem in this work, where uncertainty comes from how the environment is mapped and where pumps are located.

Mapping an unknown environment and locating where the robot is in that environment is one of the most well-known problems in mobile robotics and autonomous driving. Since localization sensors will drift over time, it makes them inaccurate when a robot has been exploring for a while. This problem is known as simultaneous localization and mapping (SLAM) [16]. This problem is crucial to solve, to allow robots to operate autonomously in an unknown environment. State-of-the-art solutions to this problem use graph-based algorithms to solve this problem, also known as graph-based SLAM [17] [5]. In graph-based SLAM the environment is represented as a graph. Nodes in the graph can be either locations of where the robot has been on the map or locations of landmarks that have been identified. Landmarks are distinctive features in the environment, which can be used as reference points for localization and mapping. The edges in the graph represent the relationship between the nodes. They can, for example, relate the position of a landmark to the position of the robot when that landmark was observed. The goal of graph-based SLAM then becomes to find the configuration of nodes and edges, that best explain what has been observed by the sensors on the robot. In other words, the algorithm tries to minimize the error between the predicted and observed measurements. State-of-the-art graph-based SLAM solutions like Slam Toolbox [5], can map rooms larger than $30.000m^2$ with a normal Intel CPU that can sit on most robots.

3. Reinforcement Learning

In reinforcement learning an agent learns optimal behaviours, by interacting with an environment [18, p. 1-5]. Unlike other machine learning paradigms that are centered around learning on data, reinforcement learning is centered around the concept of trial and error based on a model of the environment. Agents learn by taking actions and observing how those actions influence the environment. This way of learning is visualised in fig. 2, which shows that the agent interacts with the environment over a series of discrete timesteps. At every timestep t the agent selects an action a_t, that moves the environment from state s_t to s_{t+1}. A state encapsulates everything the agent knows about the environment at a given timestep t. The transition from s_t to s_{t+1}, will also result in a numerical reward r_{t+1}. This reward will indicate whether or not the action was a good choice.

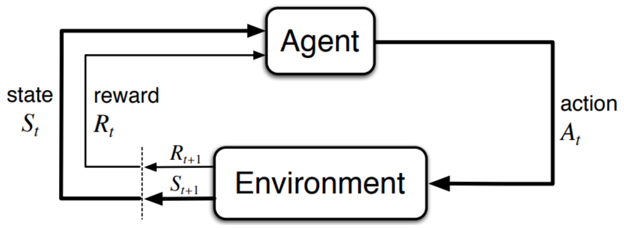

Fig. 2: The agent-environment interaction process [18]

The goal of reinforcement learning is to learn a policy that maps states to actions to maximize the expected accumulated reward [18, p. 1-3]. This section will give a theoretical overview of the components of reinforcement learning. Section 4 will present how these formulations can be used to describe the case study presented by Grundfos.

A. Markov Decision Process

The learning process shown in fig. 2 can be formulated in terms of a Markov Decision Process (MDP) [18] [19]. We first define a probability distribution function, to simplify the definition of an MDP.

Definition 1 (Probability Distribution Function): *A probability distribution function is a function $\mathcal{P} : X \to [0, 1]$ such that $\sum_{x \in X} P(x) = 1$ where X is a set of finite size. The set of all distributions on X is denoted by $Distr(X)$.*

Using the probability distribution function in definition 1, the MDP $\mathcal{M}$ can be defined.

Definition 2 (Markov Decision Process): *An MDP is a tuple $\mathcal{M} = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{T}, \mathcal{R})$:*

- *$\mathcal{S}$ is the set of finite states*
- *s_0 is the initial state*
- *$\mathcal{A}$ is the set of finite actions*
- *$\mathcal{T}$ is the probabilistic transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to Distr(\mathcal{S})$*
- *$\mathcal{R}$ is the reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$*

An example of how an MDP can be visualised is shown in fig. 3.

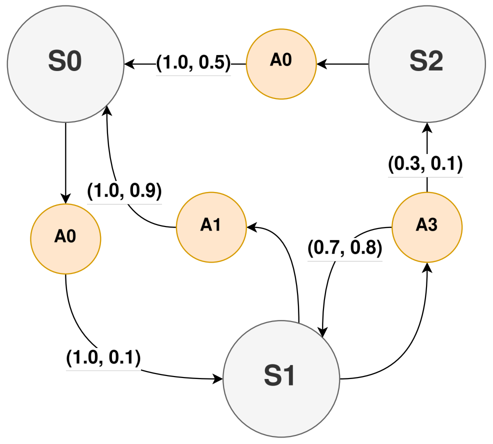

Fig. 3: Example of an MDP as defined in definition 2. The annotations on the edges are a tuple where the first element is the probability of transitioning to a given state, and the second is the reward for that transition.

Solving an MDP means finding a policy that maximizes the expected accumulated reward. There are many ways to solve an MDP, some of which will be outlined in section 3-B. An important property of an MDP is the Markov property, which states that the next state only depends on the current state and the selected action. In other words, the transition function doesn't account for the previous history of states, making solving an MDP easier.

B. Solving an MDP

The solution to an MDP is a policy that maximizes the expected accumulated reward.

Definition 3 (Expected Accumulated Reward): *We define the expected accumulated reward as $G_t \equiv \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ s.t. R_t is the reward at the timestep t.*

$\gamma \in [0, 1)$ is called a discount factor, specifying how much the agent should value future rewards. A future reward at some timestamp t is thus worth γ^{k-1} times what it would be worth if received immediately [18, p. 55-59].

Definition 4 (Policy): *A policy π is a mapping from states to actions: $\pi : \mathcal{S} \rightarrow Distr(A)$ [18, p. 55-59]*

This means that the policy π tells the agent with a probability which actions are best to take, in a given state. We denote the optimal policy π^*. One way to formally evaluate a given policy is by using a state value function [18, p. 70-75]: $v_\pi(s) = E_\pi[G_t|S_t = s]$. The function tells us the expectation of the accumulated reward, given that we are in some state and follow some policy. We can also evaluate taking a specific action in a specific state, under some policy, by using the action value function: $q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a]$. Using the state value function, we can more formally define the optimal policy π^* as the policy that yields the highest expected accumulated reward: $v^*_\pi(s) \geq v_\pi(s)$ for all $s \in S$, and any policy π [18, p. 75-79].

If we have complete knowledge of an MDP, meaning that we have the transition probability $p(s'|s,a)$ for all states $s \in S$ and all actions $a \in A$, we can compute π^* using dynamic programming [18, p. 89-98]. In other words, we need to know the probability of going to state s' and receiving reward r, if we are in state s and take action a. An algorithm that utilizes dynamic programming is policy iteration. Policy iteration is an iterative algorithm in reinforcement learning that alternates between policy evaluation, where the value function of a policy is computed, and policy improvement, where the policy is updated greedily to the current value function until it eventually either converges or is close to converging to π^*. If we don't have complete knowledge of an MDP, we can use Monte Carlo Methods or Temporal Difference learning [18, p. 143-148]. A popular variance of Temporal Difference learning is Q-learning [18, p. 157-160]. Q-Learning is also an iterative process, where an action-value function is updated based on the following update rule: $q(s_t, a_t) = q(s_t, a_t) + \alpha * (r_{t+1} + \gamma * \max_a q(s_{t+1}, a) - q(s_t, a_t))$. α is the learning rate, and γ is a discount factor.

4. The GP as an MDP

To properly define the GP as an MDP, we must have an understanding of which possibilities and restrictions the problem has. The UAV used for this project is armed with three important components: A camera, a LiDAR sensor and an odometry sensor.

- **The camera**: is used for examining pumps, using a neutral network provided by Grundfos. This is a continuous feed, where frames are sent to the neural network, meaning that there is no need for an action that tells the drone to examine a point of interest. The drone simply needs to be close enough to a POI, for the neural network to be able to detect it. In earlier work [20], we tested this neural network, and results showed that the neural network had a 100% accuracy when it was between 0.75 meters and 0.55 meters from the pump.
- **The LiDAR sensor**: is used to map the unknown environment, and to locate where in that environment the drone currently is. LiDAR uses laser technology to measure the distance to any object, the laser hits.
- **The odometry sensor**: This is used to get information about how the drone has moved from its start position (also known as its odometry). This sensor will be used in collaboration with the LiDAR sensor to help locate the drone.

Ideally, it should be possible to utilize all three dimensions that a drone can move in when working with drones. This is because objects and pumps in a room may not be possible to detect without having a dimension for height. However, in this work, we are reducing the problem to a 2D space based on assumption 1.

Assumption 1: *It is possible to project a 3-dimensional space to a 2-dimensional space, while still guaranteeing safety and creating a meaningful map. Furthermore, it is possible to detect every pump in a 2-dimensional space.*

In this project, we propose having an algorithm that can detect patterns in a map, that have some percentage of being a pump. These patterns can be seen as POI that may or may not be of value. This algorithm will not be developed in this project but is assumed to be possible to create.

Assumption 2: It is possible to create an algorithm that given a matrix can detect patterns based on the values in said matrix.

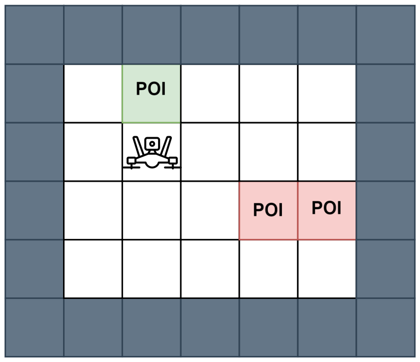

Fig. 4: Example of a map where an algorithm has detected different patterns in a map. The grey cells are walls or objects in the environment, and the white cells are free cells where the drone can freely fly. A green POI indicates a POI that has been examined, and a red POI indicates a POI that is yet to be examined.

A pattern is a coordinate in the map, that has a chance of being a pump. In terms of this work, a pattern is the same as a POI.

Definition 5 (Pattern Set): *P is a set of tuples $P = \{(x,y)|x,y \in \mathbb{N}_0\}$, where each (x,y) tuple indicates a unique point of interest for a possible pump.*

A common dilemma in RL is to balance exploration and exploitation properly. Exploration in this problem is to explore the environment and exploitation is to examine detected patterns or POI. The most important part of this project is to locate and examine all pumps. However, since there is no way of knowing the number of pumps in a given unknown environment, the drone must map almost 100% of the room to guarantee that every pump has been found. This is of cause unless the amount of pumps in a room is given beforehand, but in this work, it is assumed that the number of pumps in an environment is unknown. Another crucial factor is time since the drone is strictly limited to its battery life. Using a 5000mAh battery it can be expected that the drone can fly for around 18 minutes [21]. It is, therefore, a requirement that the drone has finished its task in less than 18 minutes.

The MDP will be given an initial state, with the current readings from the sensors and the current map. This is the position of the drone in the map and its current rotation around the yaw axis. The transitions in the MDP, as we shall see, will simulate how the components on the drone will act. For instance, the MDP will assume that moving 1 meter north always results in moving exactly 1 meter north. How such assumptions are taken care of will be explained in section 7.

A state in the GP should encapsulate the position of the drone in the map, as well as the map the drone is in.

Definition 6 (Map): *A map is a tuple $M = (m, P, g, e, c)$:*

- *m is a $h \times w$ matrix where $h, w \in \mathbb{N}_0$. Each cell in m has a value from $\{unkown, free, occupied, POI, examined\}$.*
- *P to specify which cells contain a possible pump, such that for all $(x, y) \in P$, $m_{xy} \in \{POI, examined\}$.*
- *g is the granularity of the map, i.e. $g = 0.05$ means that there are 20 cells in m for each meter.*
- *e is the number of cells in m with the value $examined$.*
- *c, where $c \in \{True, False\}$ indicates whether or not the map has been fully explored.*

As a shorthand, we write m to access m in M. Similarly, we use the same notation to access the other elements of the given map. We denote the set of all maps M as $\mathbb{M}$. At any timestep t the drone is in a state $s \in S$ where S is the set of all states. We define the set of all possible yaw directions the drone can be in as $\mathcal{Y} = \{-\frac{\pi}{2}, 0, \frac{\pi}{2}, \pi\}$. A state $s \in S$ is defined as in definition 7.

Definition 7 (State): *A state is a tuple $s = (M, x, y, yaw, t)$:*

- *M is the current map*
- *$x, y \in \mathbb{N}_0$ are row and column indexes, denoting which cell in m the drone is in.*
- *$yaw \in \mathcal{Y}$, indicating the direction the drone is currently facing.*
- *t is the current time*

We define the finite set of actions $\mathcal{A}$ as:

$$\begin{aligned}\mathcal{A} =& \{move_{x,d} \mid d \in \{-0.5, -1, 0.5, 1\}\} \cup \\ & \{move_{y,d} \mid d \in \{-0.5, -1, 0.5, 1\}\} \cup \\ & \{turn_{\Delta} \mid \Delta \in \{-\frac{\pi}{2}, \frac{\pi}{2}\}\}\end{aligned}$$

where the move actions are for moving the drone along either the row or column of a map and the turn action is for turning the drone to face a specific yaw. With this definition of $\mathcal{A}$, the drone can take 8 actions. The d value denotes the distance that the drone should move in the map in meters, so 0.5 is half a meter and 1 is one meter. These values are used to move the drone in m. For the turn action, Δ denotes the change to the drone's yaw.

The transition function $\mathcal{T}$ is deterministic, meaning there is no uncertainty about which state the drone will be in after taking an action. To help provide an understanding of the transition functions, we explain the side conditions and why they are needed. The agent's action will change the drone's position and update the map. The map is updated by an update map function Γ on the form:

$$\Gamma : \mathbb{M} \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathcal{Y} \rightarrow \mathbb{M} \tag{1}$$

The exact implementation of Γ can vary depending on the assumptions on how the map changes during exploration, as will be discussed in section 5. For now, it is enough to know that such a function is present. To formally define $\mathcal{T}$, we denote a transition function as $s \xrightarrow[\Gamma]{a} s'$, s.t. the result of the transition is dependent on a given map update function Γ. a is the action taken, Γ is the given map update function, and s and s' are the states before and after the transition, respectively. This allows us to investigate how different assumptions of the environment affect the final solution. The possibility of taking a move transition depends upon cells being free. Since the drone overlaps several cells (because of small granularity), most of the side conditions define the range of cells to validate are free. First, e is needed because even though we are saying the drone is in a single cell in m, the drone is larger than that. We use e to denote how many cells the drone covers. k denotes how many cells we are moving the drone, i.e. if the drone is to move 1 meter, k will be the number of cells 1 meter will cover. Both e and k are floored because the rows and columns in m start from 0. λ and γ are based on e and are used to give us the bounds of how large the drone is. Intuitively, λ, γ and k give us the bounds of cells that need to be $free$ for a transition to be possible, meaning that all the cells the drone covers should be free for the whole movement. The transition function for the move and turn actions can then be defined:

$$(M, x, y, yaw, t) \xrightarrow[\Gamma]{move_{x,d}} (M', x', y, yaw, t') \text{ if}$$
$$e = \left\lfloor \frac{drone\_radius}{g} \right\rfloor \text{ and}$$
$$k = \left\lfloor \frac{d}{g} \right\rfloor \text{ and}$$
$$\lambda = y - e \text{ and}$$
$$\gamma = y + e \text{ and}$$
$$\forall j \in [\lambda ... \gamma], \forall i \in [0...k] \text{ then } m_{j,x+i} = free \text{ and}$$
$$M' = \Gamma(M, x', y, yaw) \text{ and}$$
$$x' = x + k \text{ and}$$
$$t' = t + 1$$

$$(M, x, y, yaw, t) \xrightarrow[\Gamma]{move_{y,d}} (M', x, y', yaw, t') \text{ if}$$
$$e = \left\lfloor \frac{drone\_radius}{g} \right\rfloor \text{ and}$$
$$k = \left\lfloor \frac{d}{g} \right\rfloor \text{ and}$$
$$\lambda = x - e \text{ and}$$
$$\gamma = x + e \text{ and}$$
$$\forall j \in [\lambda ... \gamma], \forall i \in [0...k] \text{ then } m_{y+j,i} = free \text{ and}$$
$$M' = \Gamma(M, x, y', yaw) \text{ and}$$
$$y' = y + k \text{ and}$$
$$t' = t + 1$$

$$(M, x, y, yaw, t) \xrightarrow[\Gamma]{turn_{yaw,\Delta}} (M', x, y, yaw', t') \text{ if}$$
$$yaw' = yaw + \Delta \text{ and}$$
$$M' = \Gamma(M, x, y, yaw') \text{ and}$$
$$t' = t + 1$$

With d being the distance to move in meters, and Δ is how much to turn in radians.

We can now define the reward function $\mathcal{R}$ as:

$$\mathcal{R}(s, a, s') = r_c * \mu(s, s') + r_{poi} * \rho(s') - t' \qquad (2)$$

where

- r_c is the reward for changing a cell in m from $unkown$.
- r_{poi} is the reward for examining a POI.
- $\mu : \mathbb{M} \times \mathbb{M} \to \mathbb{Z}$, is a function that given two maps M and M', compares the cells in m and m' such that it returns the total number of cells changed from $unknown$ in m to either $free$ or $occupied$ in m'.
- $\rho : S \to \{0, 1\}$, is a function that checks if a POI is in range and can be seen by the drone in the given state. Returns 1 if a POI is in range, 0 otherwise.
- t is the current time step.

Lastly, to know when the drone has completed its job we define a goal state. Intuitively, an optimal policy π^* will reach the goal state in the shortest time possible.

Definition 8 (Goal States): *The set of goal states is* $\{(M, \_, \_, \_, \_) \in S \mid M = (\_, P, \_, e, True) \wedge |P| = e\}$. *Then the set of goals states are those states where the map is fully explored and the number of POIs is equal to the number of examined points.*

5. Uppaal Stratego

Uppaal Stratego is a solver for Stochastic Priced Timed Games (SPTG) and we are using such SPTG to encode our MDP $\mathcal{M}$ [3]. Uppaal Stratego utilizes Q-learning to find near-optimal solutions to given strategy queries [22]. A synthesized strategy, or policy, can be simulated in Uppaal Stratego, where Uppaal Stratego will simulate an MDP under a learned strategy until a goal state or stop condition has been reached. Uppaal Stratego offers a rich language for defining SPTGs, which can be used to define extended timed automatas. Timed automatas add the complexity of being able to set and test clocks. This can be beneficial because taking a transition should not be instant, since the drone can't move or turn instantly. In Uppaal Stratego such extended timed automatas are called templates. They are called extended because they can communicate and rely on each other and have a C-like language to define functions. A template consists of locations and edges. Locations can be seen as vertices in a directed graph, and edges are the connections between the locations. An edge can be either controllable or uncontrollable. Controllable edges are those that the agent can choose from, while uncontrollable edges represent effects beyond the agents' control. In Uppaal Stratego controllable edges are represented by solid edges and uncontrollable by dashed edges. An edge can have four different annotations that are all used in this work:

- **Select**: Select annotations are a way of creating copies of an edge with different values. If an edge is annotated with $x : int[1, 2]$, there will be a total of 2 versions of that edge: one where x is 1 and one where x is 2. In Uppaal Stratego select annotations are written in a yellow color.
- **Guard**: A guard is a side-effect-free expression, that always evaluates to a boolean value. If an edge is annotated with a guard, the guard must evaluate to true for the agent to take that edge. In Uppaal Stratego guard annotations are written in a green color.
- **Sync**: Sync annotations are used to allow for communication between templates. An edge can either listen to a broadcast channel or send to a broadcast channel using this annotation. $a?$ means that an edge is listening to the broadcast channel a and $a!$ means that the edge will send to the broadcast channel a. An edge that is listening to a broadcast channel will be taken if something is sent to that broadcast channel, and it is possible to take it. In

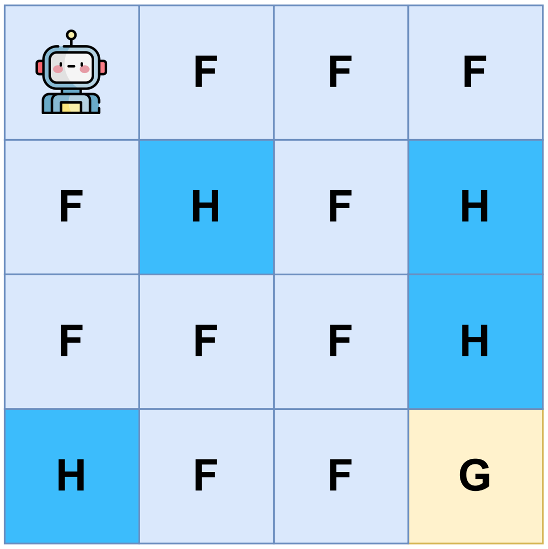

Fig. 5: Example of how a grid in the Frozen Lake problem might look. The cells annotated by F are the cells that are frozen, that the agent can safely move on, but might risk slipping. The cells annotated by H are the cells that the agent must avoid, to not fall through the ice. Lastly, the cell annotated by G is the goal cell, that the agent must reach.

UPPAAL STRATEGO sync, annotations are written in a light-blue color.

- **Update**: An update annotation is a side-effect that will be evaluated if that edge is taken. If there is a synchronization between processes, the sender's updates are evaluated first. In UPPAAL STRATEGO update annotations are written in a dark-blue color.

To illustrate the capabilities of UPPAAL STRATEGO, we shall model the Frozen Lake problem [23]. The Frozen Lake problem involves a grid representing a frozen lake, where each cell can be frozen, a hole, or the goal. The agent starts at some position in the grid and must navigate to the goal cell while avoiding the hole cells. While the frozen cells might be safe in an immediate sense, the frozen cells introduce uncertainty to the agents' movement given the slippery nature of the frozen cells. This means that moving from one frozen cell to another has a probability of moving two cells if the agent slips. For example, moving one cell to the right has some probability of moving two cells to the right. There is a total of 4 actions in this problem: move left, move right, move up and move down. An illustration of the grid that we will be modelling in UPPAAL STRATEGO can be seen in fig. 5.

To represent the grid from fig. 5 in UPPAAL STRATEGO, we construct it as a 2-dimensional array. We can then model the dynamics of the problems in UPPAAL STRATEGO as in fig. 6.

As we can see in fig. 6 the agent has a total of 4 controllable edges to choose from, which represents the 4 actions of moving. Taking any of these actions calls a function called move, which is responsible for moving the agent in the grid,

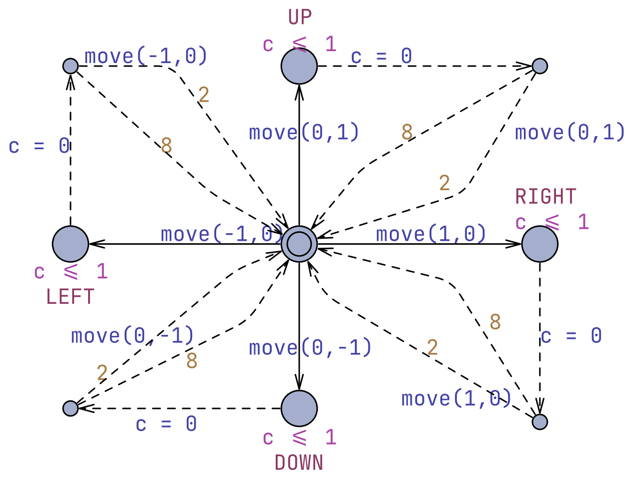

Fig. 6: The Frozen Lake problem modelled in UPPAAL STRATEGO.

as well as checking if the agent has reached the goal state or fallen into a hole. Once the agent has chosen any of the 4 edges, it is the environment's turn to choose an edge (the uncontrollable edges). The environment can choose to simply do nothing or to make the agent slip and as a result, move the agent one extra cell. There is a 20% chance of slipping, denoted by the yellow 2 in fig. 6.

Now that the dynamics of the Frozen Lake problem have been modelled in UPPAAL STRATEGO, we can use UPPAAL STRATEGO to compute strategies, verify properties, and simulate strategies. This is done through UPPAAL STRATEGO query syntax. To guarantee that the agent never is going to fall into a hole, we can compute a control strategy that makes sure that this constraint is always satisfied:

```
strategy Safe = control: A[]
                          is_dead == false
```

In the above control strategy, called Safe, we compute a strategy where the variable is_dead is always false, no matter what the environment chooses to do. This means that even if the agent is very unlucky, and slips on every move, the agent is guaranteed not to fall into any holes. This control strategy acts as a shield on the actions the agent can take. In just a millisecond UPPAAL STRATEGO has managed to compute such a strategy. However, this strategy is quite easy to achieve, since it doesn't say anything about reaching the goal. This means that the agent can simply go up and down, and still satisfy this safety constraint. To make the agent move to the goal cell, we can compute a strategy that makes the agent move to the goal in the minimum amount of time, while still satisfying the safety constraint. A strategy query consists of three important elements:

- **Minimize or maximize**: The first part of the strategy is used to tell UPPAAL STRATEGO what value to minimize or maximize. In this case, we want to minimize the time taken to reach the goal.

- **Discrete and continuous variables**: To tell the agent which variables are observable we use the $\{...\} \rightarrow \{...\}$ syntax. The discrete variables are placed in the first bracket. Discrete variables are variables of discrete types such as *int* and *boolean*. Discrete variables are observed as they are, i.e. the value 10 is observed as 10. However, continuous variables (variables of continuous types such as *float*), are not observed as they are, since they can be infinitely precise. Instead, they are discretized using online partition refinement [3]. This means that instead of observing some variable x as being exactly 12.6, it might be observed as falling within a certain interval, such as being within the interval [12.6, 12.7].
- **Stop condition**: The stop condition tells UPPAAL STRATEGO when to stop a specific simulation in the training session. In our case, a specific simulation stops after reaching the goal state, meaning that the variable *has_reached_goal* is true.

We can now construct the learning query:

```
strategy SafeFast =
        minE(time) [<=100]
         {agent_x, agent_y} -> {} :
         <> has_reached_goal
        under Safe
```

What the above query means, is that we try to find a strategy where *has_reached_goal* is true in the minimum amount of time, under the control strategy Safe. This strategy can be computed in around 5 seconds. There are different ways of evaluating these strategies, we can, for instance, use a simulate query to see the number of times we reach the goal state:

```
simulate[<=10; 100000]
        {agent_x, agent_y}:
        has_reached_goal
        under SafeFast
```

The above simulate query tells us that out of 100.000 runs, we reach the goal state 100% of the time in less than 10 time units. We can also evaluate the safety:

```
A[] is_dead == false under Safe
```

The above query checks that it is always satisfied that the *is_dead* variable is false, given the control strategy Safe. This query tells us that this is indeed satisfied.

6. THE GP IN UPPAAL STRATEGO

The GP can be encoded as an SPTG using UPPAAL STRATEGO, in a similar way that the frozen lake problem was encoded as an SPTG using UPPAAL STRATEGO. Just like in the frozen lake problem, we can define the map as a 2-dimensional

Algorithm 1 *Updating M*

```
1: Input: $s_t = (M, x, y, yaw, t) \in S$, laser_range, laser_diameter, upper_range_detection, lower_range_detection.
2: Output: An updated map $M$.

3: $forward\_cells \leftarrow floor(laser\_range/g)$
4: $diameter\_cells \leftarrow floor(laser\_diameter/g)$

5: if $forward\_cells \mod 2 = 0$ then
6:     $foward\_cells += 1$
7: end if
8: if $diameter\_cells \mod 2 = 0$ then
9:     $diameter\_cells += 1$
10: end if

11: // DRONE IS FACING POSITIVE Y-DIRECTION
12: if $-\frac{\pi}{2} - 0.2 < yaw \;\&\; yaw < -\frac{\pi}{2} + 0.2$ then
13:     $lower\_bound\_x \leftarrow x - (diameter\_cells/2)$
14:     $upper\_bound\_x \leftarrow x + (diameter\_cells/2)$
15:     $upper\_bound\_y \leftarrow y + (foward\_cells/2)$
16:     if $lower\_bound\_x < 0$ then
17:         $lower\_bound\_x \leftarrow 0$
18:     end if
19:     if $upper\_bound\_x > map\_width$ then
20:         $upper\_bound\_x \leftarrow map\_width$
21:     end if
22:     if $upper\_bound\_y > map\_height$ then
23:         $upper\_bound\_y \leftarrow map\_height$
24:     end if
25:     for $i$ in range$(lower\_bound\_x, upper\_bound\_x)$ do
26:         for $j$ in range $(y + 1, upper\_bound\_y)$ do
27:             if $m_{j,i} = free$ then
28:                 $j \leftarrow upper\_bound\_y$
29:             end if
30:             if $(j, i) \in P \;\&$ drone is nearby then
31:                 $n \leftarrow n + 1$
32:             end if
33:             if $m_{j,i} = unkown$ then
34:                 if $open = False$ then
35:                     $m_{j,i} \leftarrow occupied$
36:                 end if
37:                 if $open = True$ then
38:                     $m_{j,i} \leftarrow free$
39:                 end if
40:             end if
41:         end for
42:     end for
43: end if
44: // There are 3 more if statements, like the one on line 12 for the other directions that the drone can be facing.
45: return $M$
```

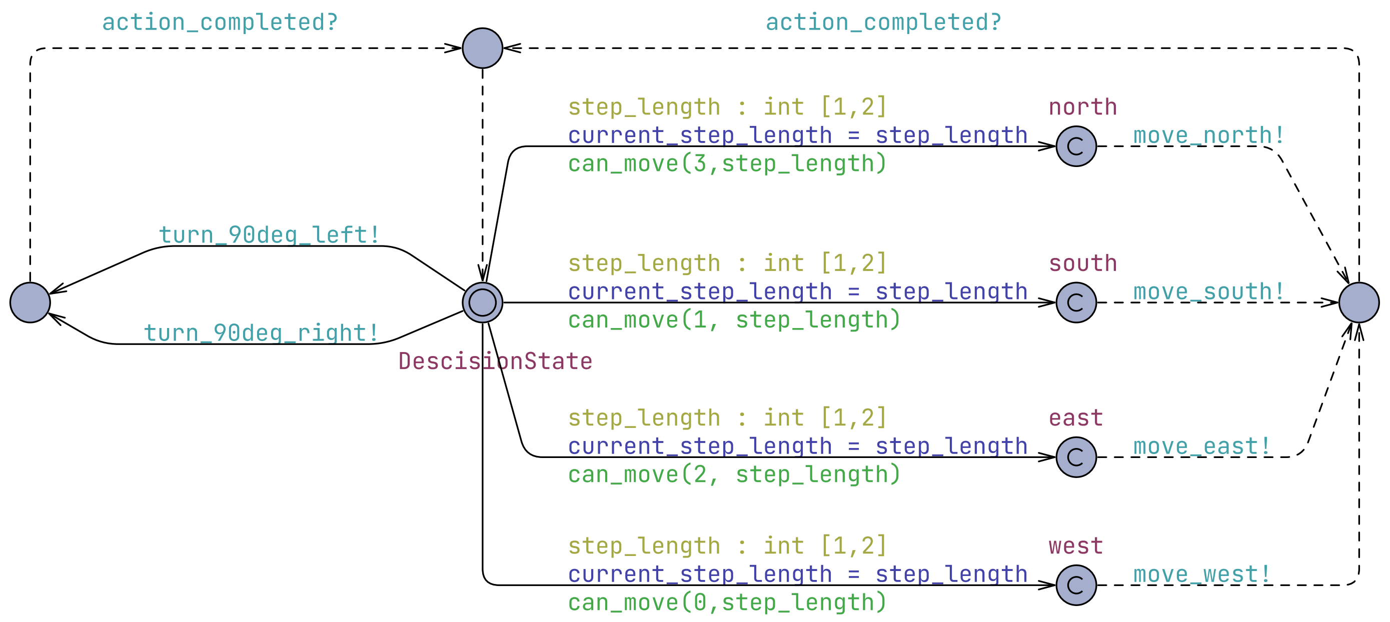

Fig. 7: UPPAAL STRATEGO Controller Template

matrix, only this time, the cells can have a total of 5 different values $(unkown, free, occupied, POI, examined)$ as defined in definition 6. As we shall see in this section, the GP problems also introduce more complexities such as updating the map when moving and guaranteeing safety.

In this work, we have formulated a total of 3 templates in UPPAAL STRATEGO: A controller, a template for moving, and a template for turning. The reason for having three templates and not only one, is to make the system more modular. For instance, we can modify the moving template if we want to modify the moving functionality, without having to modify anything else. The controller template, which can be seen in fig. 7, is the only template with controllable edges. When the agent is in the location "DescisionState", it can choose between the 10 actions that were outlined in section 4. The turn edges are always available for the agent, but the move edges may be blocked by a guard. The guards that call a function can_move, check if it is safe to take that action with respect to the current map M. These guards can also be seen as a form of shield [24]. A shield is a system, that will correct any action given to it, that does not satisfy a safety constraint. Of course guards in UPPAAL STRATEGO won't correct actions, they will simply not allow the agent to take those actions. In this paper we formally define our only safety specification ϕ, using Linear Temporal Logic.

Definition 9 (Safety Specification): *We define our safety specification as $\phi = G(d > \epsilon)$, where G means invariant, d is the distance to the closest object and ϵ is a value, indicating how close the drone is allowed to be in meters. A safe action is thus any action where after it has been executed, d is still greater than ϵ.*

The moving actions are the only actions that can change the distance d, thus they are the only actions with the can_move guard annotation. can_move is thus a function that evaluates to true if ϕ is satisfied after the agent has taken some action. Looking at fig. 7 it can be seen how every controllable edge, eventually will result in sending a message to a broadcast channel. This will make the other templates evaluate that action, just like the transition functions outlined in section 4. The moving and turning templates are shown in appendix A and appendix B, respectively. After sending a message to some broadcast channel, the controller template will wait for one of the other templates to send a message to the broadcast channel $action\_completed$. This message will be sent once the other template is done evaluating the action. The moving template and the turning template will call a function called $calculate\_reward$ that adds a value to the accumulated reward, based on the reward function outlined in section 4.

Algorithm 1 is called after each action, and will update the map M. This algorithm is the algorithm we use for Γ in our transition functions. The algorithm is an attempt to simulate how the map M would normally change, using an LiDAR sensor. The algorithm takes a total of 6 parameters:

- s_t the current state.
- $laser\_range$ the range of the LiDAR sensor in meters.
- $laser\_diameter$ the diameter of the LiDAR sensor in meters.
- $upper\_range\_detection$ the max distance a drone can be from a POI to examine it.
- $lower\_range\_detection$ the min distance a drone can from a POI to examine it.

The algorithm looks at the cells in m that are in the range and radius of the laser and are in the direction the drone is facing. Depending on the value of the cell the algorithm will act differently. If the cell is occupied, the algorithm will stop

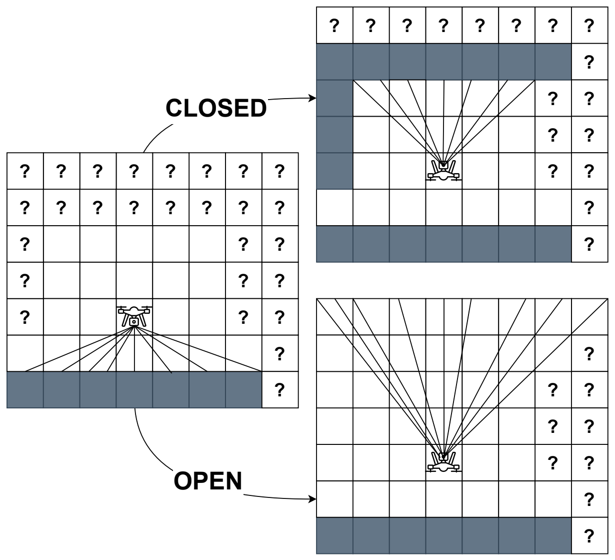

Fig. 8: Example of how the map updates after taking the action ”turn 90 degrees” twice, with the respective mapping modes ”open” and ”closed”. Grey cells symbolise walls or unsafe cells, and cells with a question mark indicate unknown cells. The consequence of using the open mapping mode, is that the reward increases.

looking at that row. If the cell is unknown, it will change it to either free or occupied, based on a configuration called *open* or *closed*. This configuration is made so that we can test different approaches for UPPAAL STRATEGO to try and predict how the map changes. If the algorithm is configured to be *open* it will change the unknown cell to a free cell, otherwise, it will change it to an occupied cell. A visualisation of the differences between the two configurations can be seen in fig. 8. If a cell is in the range for detection and has the value POI, the value of the cell will change to $examined$ and e will be incremented by 1.

To learn a strategy, we use the queries outlined in table I. We want to maximize the accumulated reward subtracted by the time taken. The h in this case is the horizon. That is the maximum amount of actions we are calculating. For instance if $h = 20$, we want to find the sequence of the next 20 actions, that maximizes the reward. The reason for doing this, and not simply calculating the strategy for reaching the goal state, is that due to the size of the state space, the learning time would be too great. As an example, given a granularity of 0.05 and a $100m^2$ room, the map will have a total of 40.000 cells when fully explored. The only state variable is *DroneController.DescissionState*, which is the state in which the agent can chose an action. The point variables are x, y and yaw, which are the position of the drone and it's rotation. Once a strategy has been learned it can be simulated directly by using the simulate query shown in table I, or it can be saved as a JSON file. In our work, we will be simulating the query, to compute a plan from UPPAAL STRATEGO instead of saving the strategy as a JSON file.

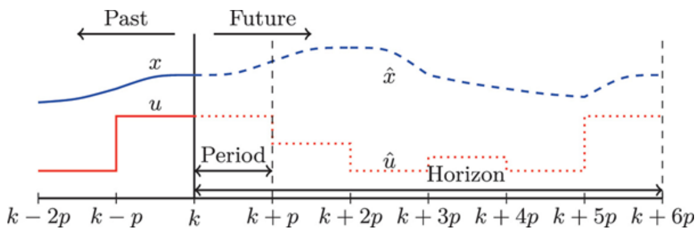

Fig. 9: Model-Predictive Control conceptual sketch [9]

7. STOCHASTIC MODEL PREDICTIVE CONTROL

When UPPAAL STRATEGO is synthesizing plans for the drone, everything during learning is deterministic. There is no stochasticity in which state the drone will be in after taking an action or following a plan. However, stochasticity is introduced into the system when the drone executes the actions in a simulation or the real world. The reason for this is the drone's behaviour, such as hovering in place which is not a static motion, or other parameters that can affect the drone, such as wind. An example of this stochasticity is telling the drone to move 1 meter north, where this might result in the drone moving 1.2 meters north, 0.8 meters north, or not north at all. Furthermore, updating map m in UPPAAL STRATEGO is done in what can be called a naive manner, since UPPAAL STRATEGO can not see which cells are unoccupied or occupied. We attempt to handle this stochastic behaviour by implementing a Stochastic Model Predictive Control scheme [25]. While such schemes fit well into UPPAAL STRATEGO it is currently not possible to actively update the current state, and learn a new strategy. Because of this the framework STOMPC [9] has been integrated into the pipeline, to form a learning loop. The architecture of STOMPC consists of three main components: A simulator, the model predictive controller, and UPPAAL STRATEGO.

Figure 9 shows a conceptual overview of how STOMPC works. Up to timestep $t = k$ observations of the true state x are made, and which control inputs u affected the state. Within some horizon, we try to predict the future state $\hat{x}$. The future state $\hat{x}$ depends on the applied control input $\hat{u}$, that is applied in every period p. In our work, a period is the time it takes for the drone to complete an action, and the control input is the action itself.

The goal of STOMPC is to have strategies ready for the many different states the system can be in after having executed an action. The authors of STOMPC propose that when UPPAAL STRATEGO has determined the optimal control sequence, based on the observed true state, we only execute the first control input in that sequence. While the action is being executed, UPPAAL STRATEGO should compute a new optimal control sequence based on the new observed true state. However, because the map is continuously updating and expanding while the drone is executing actions, this is potentially not the best approach. This is because UPPAAL STRATEGO has

Purpose	Query
Learn a strategy that maximizes accumulated reward	strategy opt = maxE(accum_reward - time) [<=1000] {DroneController.DescissionState) -> {x, y, yaw} : <>(time >= h)
Simulate the first h actions of the learned strategy opt	simulate [$\leq$1000;1] {action} : (time $\geq$ h) under opt

TABLE I: The main queries that can be run either through the UPPAAL STRATEGO user interface, or through the command line using UPPAAL STRATEGO verifyta.

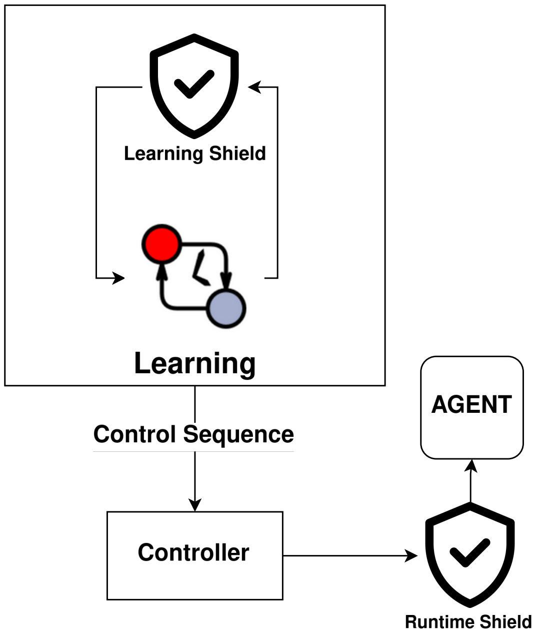

Fig. 10: Where the two shields are deployed in the system.

no way to predict this expansion and can only naively make predictions about the cell updates. This expansion is described in section 9-C. Instead, we execute every action in the control sequence or all actions until an unsafe action is reached. After executing all safe actions in the control sequence, a new control sequence from UPPAAL STRATEGO is requested based on the observed new true state. We can define the control sequence as in definition 10.

Definition 10 (Control Sequence): *A control sequence* $\theta = \{a_i | a_i \in \mathcal{A} \wedge 0 \leq i < h\}$ *is the set of actions* i *from* $i = 0$ *up to* $i = h$*, where* h *is the number of actions to get from the simulate query and* $i, h \in \mathbb{N}_0$.

8. SHIELDING

In this work, we propose having two different shields, which fundamentally have the same functionality but are placed differently and, therefore, act at different points in time. The shield in UPPAAL STRATEGO will be called the learning shield, while the shield in STOMPC will be called the runtime shield. The learning shield is active during learning in UPPAAL STRATEGO, so that UPPAAL STRATEGO can't execute any action that is unsafe w.r.t to its abstract state. However, since UPPAAL STRATEGO can only guess how the map evolves, it can never guarantee safety. Thus the runtime shield is deployed in STOMPC, to make sure that the action is also safe w.r.t to the true state of how the map has evolved. The benefit of having two shields is that we can execute several actions from the action sequence θ given by UPPAAL STRATEGO. STOMPC will execute each action $a \in \theta$ one at a time. The first action a_t from UPPAAL STRATEGO is guaranteed to be safe, given that UPPAAL STRATEGO in timestep 0 has the true state of the map m. When the action is done executing, STOMPC will get the new true state of the map m, and shield the next action a_{t+1}. STOMPC will keep doing this until it has either executed every action in θ, or the shield blocks an action. If the shield blocks an action, STOMPC will send the new true state to UPPAAL STRATEGO and request a new strategy.

9. PIPELINE ARCHITECTURE

To realise the plan computed by UPPAAL STRATEGO, we must be able to execute the actions on an actual drone. Furthermore, the true state sent to UPPAAL STRATEGO through the STOMPC framework must be retrieved from the sensors on the drone. We propose the pipeline visualised in fig. 11, that combines the use of UPPAAL STRATEGO, STOMPC, ROS, Slam Toolbox and the middleware needed to control the drone. In this section, we will first give an overview of the components that are yet to be explained (Slam Toolbox, ROS, Middleware, Environment) and then in section 10 give an example of how the entire pipeline works.

A. Robot Operating System (ROS)

ROS is a flexible and open-source framework for building software for robots [4]. The framework provides a set of tools and libraries to simplify the process of creating complex robot software. The flexibility of ROS makes it easy to switch out different components and reuse them on different robots.

Two fundamental concepts in ROS, are nodes and topics respectively. A node in ROS is a specific process, that is responsible for a specific task. A task could be processing data from a sensor, or making the drone move. Topics are one way that nodes can communicate with each other. A node can either subscribe or publish to a topic. If a topic publishes data to a topic, any node that subscribes to that topic will receive the data. In fig. 11 topics are visualised as yellow circles.

B. Middleware

In terms of this work, middleware is the software that sits between the low-level firmware on the robot and ROS. The

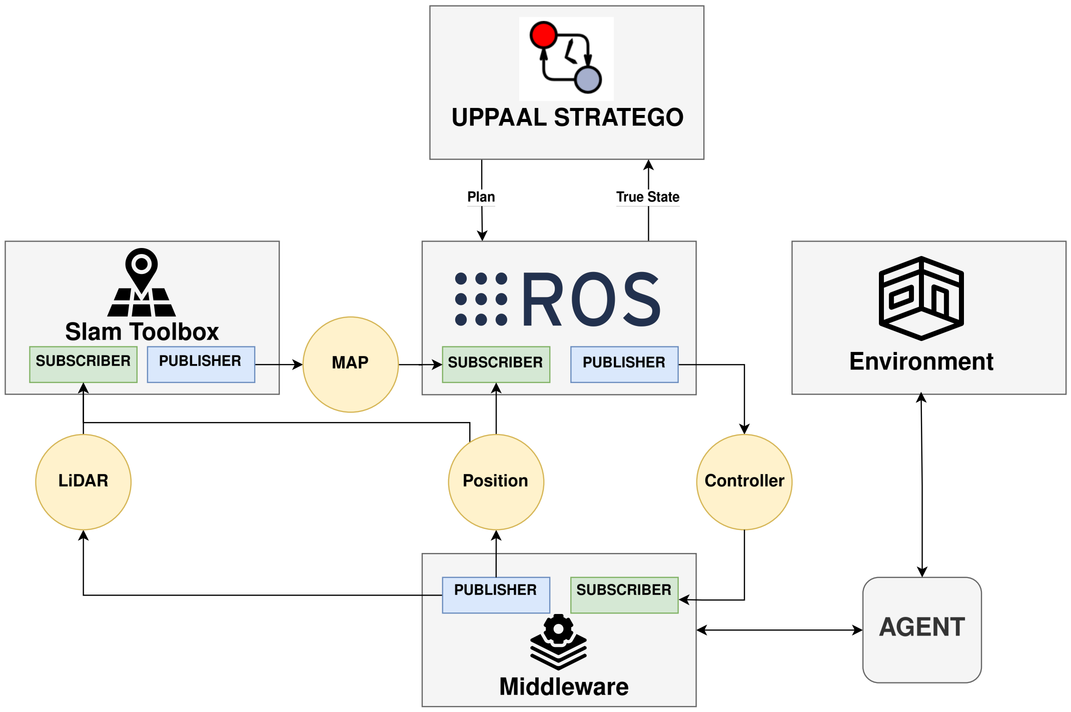

Fig. 11: Overview of the proposed architecture. Yellow circles are ROS topics, that are published or subscribed to by ROS nodes.

specific middleware is not important and can be changed, as long as it exposes three types of topics: a LiDAR topic for LiDAR data, a position topic for odometry data (position and yaw of the robot) and a controller topic for controlling the robot. As we shall see in the experiments we will be using two different middleware, one for controlling a drone in a simulation environment and one for controlling a robot in real life.

C. Slam Toolbox

Slam Toolbox [5] is used to build m. The reason for choosing this framework is that it is compatible with ROS2, well-documented, and shown to produce great results in similar cases. Slam Toolbox needs LiDAR data to create the map. The benefit of using SLAM is that not only does it provide us with a map, but also tells us where in the map the drone currently is. This is crucial since solely relying on the odometry data, is known to be inaccurate due to the drift of the drone and will worsen the longer the drone operates. Such inaccuracy will not only make it difficult to build an accurate map but also to guarantee safety constraints. When using Slam Toolbox, the map will increase in size over time. An example of this can be seen in fig. 12 where the amount of cells is increased by 8 after 2 turning actions. It is important to note that the map in the MDP that is encoded in UPPAAL STRATEGO, can not increase in size. In that map, it is only possible to change the value of a cell. However, after executing the plan from UPPAAL STRATEGO, STOMPC will create a new instance of the MDP with the larger map.

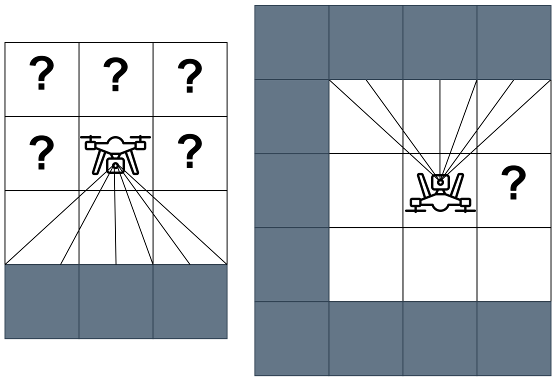

Fig. 12: Example of how the map might evolve after taking the action "turn 90 degrees" twice. Grey cells symbolise walls or occupied cells, and cells with a question mark indicate unknown cells. The lines from the camera on the drone are the laser scans from the LiDAR.

In this work, patterns will not be in P before the cells they occupy are in m, meaning that the drone should have explored enough of the room and theoretically have seen the location of the POI. This is done so that only information the drone has gathered is used for training and no outside knowledge is affecting it.

D. Environment

The environment is the physical space that the agent operates in. As we shall see in the experiments we will be showcasing the pipeline in both a simulated environment (using Gazebo) and a real-life environment. The sensors on the agent are directly influenced by the environment and are read by the middleware which finally publishes this data as ROS topics.

10. Showcasing the pipeline

In the following section, we will be giving an example of how the pipeline works, using a 3D environment simulated in Gazebo.

- **Step 1 - Getting the initial state:** STOMPC sends the initial true state into Uppaal Stratego. This includes the current knowledge of how the environment looks, the drone's position in that map and its current yaw. This data is gathered from Slam Toolbox and the published topics from the middleware.
- **Step 2 - Computing an action sequence:** Uppaal Stratego calculates an action sequence, given the current true state. Often, the first action sequence from Uppaal Stratego is to keep turning, in order to get more information about how the environment looks.
- **Step 3 - Actuating the action sequence:** STOMPC actuates each action in the given sequence one at a time. Between each action, the true state of the system will be updated, such as an increased knowledge of the environment. Because of this, STOMPC will deploy the runtime shield, between each action, making sure that the action is still safe concerning the updated knowledge. If at any time the runtime shield blocks an action, STOMPC will actuate a 360-turn on the drone, and then check if the runtime shield is still blocking the action. If the runtime shield still blocks the action, STOMPC will send the current true state to Uppaal Stratego and wait for a new sequence.
- **Step 4 - Repeat until goal state:** The final step is to keep repeating steps 2 and 3 until eventually all pumps are found and the map is fully explored.

11. Experiments

To evaluate the proposed method, we are using the Gazebo Simulator [7] as the simulator for STOMPC, which allows us to simulate a physical environment. Additionally, it will enable us to simulate a physical real-world drone with stochastic behaviour. For instance, hovering in place is not a static position and moving the drone from a start position multiple times with the same velocity does not entail the drone moving to a deterministic position. This means that while our Uppaal Stratego model is deterministic w.r.t how the drone moves while synthesizing a plan, Gazebo introduces stochasticity into our proposed method, given the stochastic nature of how a drone behaves in real life. The experiments done for comparison are all using the room model shown in fig. 13. From this point, we will use room to denote the 3D model in Gazebo.

For comparison, we have implemented as a baseline a Breadth-first-search (BFS) approach in Python. This approach

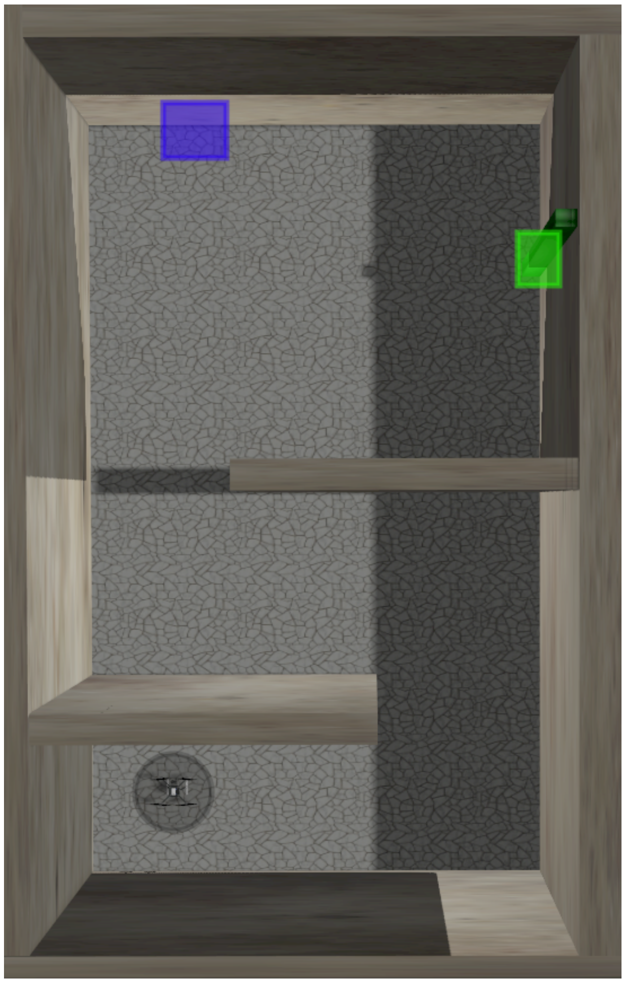
Fig. 13: The model used for all the experiments. The two squares represent POIs. The blue square indicates a potential POI which ultimately is not a pump, and the green square indicates another POI, which is an actual pump. The green pillar at the green square is used to show a pattern in the map generated by Slam Toolbox, even though they are not technically used. The black circle shows the drone's starting position before takeoff. With a granularity of 0.05 the room consist of a total of 24.480 cells.

tries to solve the problem of exploration and finding POIs by using BFS to find the closest unknown cell in the map with a safe path to it, based on the current map. If there is a POI in the map it will compute a safe path to that instead. This approach does not use any RL and relies solely on the BFS algorithm. However, it still utilizes everything in the pipeline in fig. 11, including the run-time shield. If the run-time shield ever corrects the algorithm, it will make a 360-degree turn and then compute a new path using the new map.

For all experiments, the drone has the same initial starting position. When a test is begun, the drone will perform takeoff, turning to an initial yaw value, then turning 4 times before reaching the initial yaw again. This is done to create an initial view of the room. After takeoff, both approaches will begin

Setup	Completed Task	Suspected Crashes	Avg. Completion Time(minutes)	Avg. Number of Times Trained	Avg. training time(seconds)	Avg. Number of actions
UC-M1-H20	97.89%	5	5.48 (±2.74)	13.13 (±7.73)	3.96 (±0.25)	111.03 (±67.59)
UC-M3-H10	97.89%	6	5.59 (±3.24)	11.57 (±7.40)	6.02 (±0.41)	94.11 (±65.04)
UC-M3-H20	98.96%	4	6.02 (±2.17)	10.83 (±4.56)	11.55 (±0.75)	86.48 (±35.24)
UC-M3-H30	95.55%	10	7.13 (±3.04)	10.49 (±4.70)	17.02 (±0.88)	82.29 (±37.37)
UO-M3-H10	87.50%	12	8.34 (±4.96)	20.10 (±14.28)	6.21 (±0.59)	168.24 (±129.20)
UO-M3-H20	79.55%	12	10.51 (±4.99)	19.5 (±10.94)	12.26 (±0.88)	163.73 (±96.90)
BFS-SPA	96.84%	6	4.13 (±2.72)	30.61 (±51.77)	0.04 (±0.01)	87.37 (±88.90)

TABLE II: The results from individual experiments. The data is shown with the suspected crashes removed, meaning if there are 10 suspected crashes, and out of the 90 runs left, there were 45 completed, the results would be 50%. The **avg. completion time** is shown in minutes, with **avg. training time** in seconds. The number in the parentheses is the standard deviation.

the process of exploring the room and trying to find the POIs. For each test configuration, we collect 100 runs. The goal of a run is to find all POIs in the room, of which there are two (though this number is not known to the drone controller), and have the map fully explored. To say that a map is fully explored, the map produced by Slam Toolbox must have no more than 20 consecutive unknown cells adjacent to a free cell. Intuitively, this means there must not be an opening in the map larger than 1 meter along any wall. We are subsequently calling the act of attempting to reach this goal the drone task. In the experiments, we define a suspected crash, as the room coverage being greater than some threshold. This is because when the drone flies into a wall, the SLAM map tends to explode in size. This is why the crash is only suspected, since there can be other reasons for the SLAM map to be larger than expected, such as failures in the LiDAR readings. We say that we suspect a crash if the coverage of the run is above 105%. We measure coverage by counting how many cells in m are $free$ and compare them to how many cells we expect there to be in the room.

We compare the different experiment configurations on several different variables:

1) *Total number of successful runs:* a run is successful when the drone has completed its task and is not a suspected crash. If the task is not finished after 18 minutes have passed, we say that the task has failed.
2) *Average Completion time:* the average time taken for the drone to complete the task, in minutes. This is capped at 18 minutes, due to the battery restrictions of the drone described earlier.
3) *Average number of times trained:* the average number of times UPPAAL STRATEGO has synthesized a plan or the BFS approach has to find a new path.
4) *Average training time in* UPPAAL STRATEGO*:* the average time taken for UPPAAL STRATEGO to synthesize a new plan for the drone in seconds or how long the BFS approach took to find a safe path.
5) *Average number of actions activated:* the average number of actions activated by the drone from when the experiment began to the task is either completed or 18 minutes have passed.
6) *Total number of suspected crashes:* the number of runs that we suspect have had a crash.

Our experiments are divided into three parts. For the first part, we aim to study how different combinations of $open$ and $closed$ mapping configurations, as shown in fig. 8, and different horizons in UPPAAL STRATEGO are impacting the results. These results are compared to the BFS approach. Then we show that the method proposed can be used for various models with varying sizes and shapes. Lastly, we show as a proof of concept the proposed method working on a real-world Turtlebot3 robot, where it maps a small room based on fig. 13. All experiments were done on a machine running Ubuntu 22.04 LTS with an Intel I9 9900K CPU, RTX 2700 GPU and 16GB of memory. For the proof of concept showcase, a Thinkpad T14s with an AMD Ryzen 7 PRO 4750U CPU and 32GB of memory was used.

We want to make it clear for the reader, that the experiments in section 11-A were all done with a bug in the UPPAAL STRATEGO model. We have left our findings in because those are what led us to discover the bug. We invite the reader to go to section 11-B for experiments without the bug.

An example of the drone finding all the pumps using the pipeline can be seen on YouTube[a].
A video of the TurtleBot3 experiment can be seen on YouTube[b].

[a]https://www.youtube.com/watch?v=9oKTYK9Lk-s
[b]https://www.youtube.com/watch?v=zwRT9ZsrhSI

A. Testing different configurations

To get a better understanding of how the different configurations are affecting the different comparison variables presented previously, we conduct experiments on different setups for mapping configurations and horizons. This is done to evaluate the impact of having an $open$ or $closed$ mapping configuration and to empirically determine if there are any bounds on the horizon UPPAAL STRATEGO uses to synthesize a plan, where no further gain is achievable. Finally, we will compare the results to the BFS approach explained earlier. All experiments are done in the Gazebo simulator, using the X500 drone [21] from Holybro, shown in fig. 14 with PX4 [26] as the middleware to control the drone via ROS.

All of the experiments done in this section are conducted the same way. We collect 100 runs from each of the configurations. For all the comparison variables where we compare the runs, we have removed the runs that we suspect have had crashes. All experiments are done in the environment shown in fig. 13 and we have calculated that 24.480 cells are expected to

Fig. 14: The X500 drone from Holybro [21].

be $free$. We are using the default learning parameters in UPPAAL STRATEGO except for max-iterations. Max-iterations is the total number of iterations UPPAAL STRATEGO uses in Q-learning. To denote different experiment setups using our proposed method, we use the following notation: say the experiment is done with a *open* mapping configuration in UPPAAL STRATEGO, using a max-iterations of 1 and a horizon of 20, this will be denoted as UO-M1-H20. If the experiment were using a *closed* mapping configuration, with the same max-iterations and horizon, it would be UC-M1-H20. We are calling the BFS approach for Breadth-First-Search Safe Path Approach (BFS-SPA). The results from our experiments can be seen in table II.

In regards to which mapping configuration seems to be performing best, it seems there is a clear gain in having a *closed* mapping configuration. On all comparison variables, *closed* outperforms the *open* mapping configuration, from having fewer crashes, fewer times trained, and faster completion time while also having a better completion rate. The worst performing setup with *closed* mapping, UC-M3-H30, still has over 8% better completion rate compared to the best performing *open* mapping setup, UO-M3-H10. However, the room used for the experiments is comprised of narrow corridors, where one can suspect the *open* configuration can not be utilized to its fullest. This thought requires more experiments to be fully explored, but we suspect this might be the case, with *open* configuration being more suited to wide and open-spaced rooms.

Additionally, it seems that having a longer and longer horizon in UPPAAL STRATEGO does not yield better and better results. We can see that between UC-M3-H10 to UC-M3-H20, there is a completion rate gain of about 1%. Contrarily, between UC-M3-H20 and UC-M3-H30, there is a loss in completion rate of about 3.41%. This is interesting and indicates there is an upper bound on how far UPPAAL STRATEGO can predict the map. Again, one has to remember the narrowness of the room. One can also imagine that if the room is fully explored, having a larger horizon might perform better, because the map does not change anymore. Additional experiments could go into implementing a dynamic horizon, based on how much of the map has been explored. The results also show that there is a significant gain in average training time if we are using a max-iteration of 1. If we compare UC-M1-H20 to the rest of the setups using *closed*, we can see that it is at least 2 seconds faster at synthesizing a new plan. However, this comes at the cost of taking at least 16 more actions compared to UC-M3-H10 setups, which is the setup taking the second most actions. We can also see that UC-M1-H20 trains at least 2 more times than the setup with the next most actions activated. These observations make for an interesting follow-up experiment, to try and examine the effect training and activating actions have on the battery consumption [27].

If we compare our approach to the baseline BFS-SPA, we see that it performs better on almost all parameters, with our approach being marginally better on the number of actions activated. This led us to question why the strategy synthesised in our approach was not performing better, given the long training times, than the data suggested. Based on these results and that question, we investigate our approach's possible optimizations, potential bugs and solutions to these, which will be presented in section 11-B.

B. Follow-up experiments

Our investigation into possible optimizations to our approach revealed that the strategy calculated by UPPAAL STRATEGO would give us odd control sequences, such as going back and forth repeatedly or only activate turning actions. This led us to discover a bug in the way we simulated the LiDAR sensor in UPPAAL STRATEGO. The bug caused UPPAAL STRATEGO to be able to see through walls when calculating rewards, thus being able to get a reward for removing *unknown* cells it should not be able to. This explains the odd control sequences, because UPPAAL STRATEGO thought it was updating cells, but in the simulator, it was not updating anything. This bug has been fixed in the new experiments but is present in all experiments using our approach in section 11-A. Another thing we noticed with the control sequences is that it looked to be prioritizing the shorter move actions, even though the longer move action would be better. For example, it was not uncommon for UPPAAL STRATEGO to make a control sequence where the drone moved 2 meters by using four shorter move actions, instead of using two longer move actions. In an attempt to combat this, we are giving a penalty for using the shorter actions, thus trying to limit their usage. We conduct two new experiments to try and understand the effect of the bug fix and the penalty to shorter moving actions. For the first, we are running configurations UC-M3-H20 and UO-M3-H20 again, with the only changes being the bug fix and penalty. We are calling these UC-M3-H20-FIX and UO-M3-H20-FIX respectively. This is done because we want to understand if the ability to see through walls had an impact on the *open* mapping configuration since it had to change more cells. For the second experiment, we are testing with different UPPAAL STRATEGO learning parameters, to see if there is a gain to making training time faster, at the cost of making a worse strategy. We are calling this run UO-M2-H20-ND. In this test, we are reducing the values of the following learning parameters by half of the default values: *reset-no-better*, *good-runs*, *total-runs*. We collect 10 runs for each of the configurations. Results are shown in table III.

If we compare the results of configurations UC-M3-H20-FIX and UO-M3-H20-FIX to their respective configurations

Setup	Completed Task	Suspected Crashes	Avg. Completion Time(minutes)	Avg. Number of Times Trained	Avg. training time(seconds)	Avg. Number of actions
UC-M3-H20-FIX	88.89%	1	4.94 (±2.0)	8 (±3.80)	13.46 (±0.74)	69.66 (±36.48)
UO-M3-H20-FIX	100%	0	4.60 (±0.91)	7.1 (±1.66)	14.44 (±0.37)	54.4 (±13.92)
UO-M2-H20-ND	100%	0	4.33 (±0.90)	9.10 (±2.55)	6.22 (±0.22)	75.6 (±24.56)
BFS-SPA	96.84%	6	4.13 (±2.72)	30.61 (±51.77)	0.04 (±0.01)	87.37 (±88.90)

TABLE III: The results from individual experiments. The data is shown with the suspected crashes removed, meaning if there are 10 suspected crashes, and out of the 90 runs left, there were 45 completed, the results would be 50%. The **avg. completion time** is shown in minutes, with **avg. training time** in seconds. The number in the parentheses is the standard deviation.

from the first results, we see significant improvements. Completion time is improved by over a minute from UC-M3-H20 to UC-M3-H20-FIX, with UO-M3-H20-FIX being almost a full 6 minutes faster. This result seems to imply that the *open* mapping configuration is the best but got severely punished by the bug in the UPPAAL STRATEGO model present in the previous experiments. The results also show significant improvements in the number of actions activated. UC-M3-H20-FIX took on average 18 fewer actions than UC-M3-H20, with UO-M3-H20-FIX taking 113.84 fewer actions than UO-M3-H20. We can also see that both UC-M3-H20-FIX and UO-M3-H20-FIX now activates fewer actions than the baseline. The number of times both configurations had to train for a new strategy is also lower than their counterparts from the first results, with the training time being marginally longer. If training time was not a factor, we can see that our approach is significantly better than BFS-SPA, with for example UO-M3-H20-FIX being on average 1.24 minutes faster.

We also conducted a small experiment on how the size of the map affects the training time. As fig. 15 shows, the size of the map increases the training time by upwards of 7 seconds in the room from fig. 13. This is an important factor since all of our experiments show that training time is a big part of the overall completion time. If it is possible to optimize the UPPAAL STRATEGO model to make training faster, it seems probable that our approach is competitive to the baseline. UO-M3-H20-ND in table III shows a small experiment to reduce training time by giving UPPAAL STRATEGO less time to synthesize strategies. We see that UO-M3-H20-ND is synthesizing a new strategy 8 seconds faster on average, compared to UO-M3-H20-FIX. However, this only results in an improved average completion time of 0.27 minutes. The reason for this is that the strategy produced with UO-M3-H20-ND is so much worse that it has to activate on average 21 more actions compared to UO-M3-H20-FIX. This highlights the importance of optimizing the UPPAAL STRATEGO model to improve training time, instead of lowering learning parameters to make UPPAAL STRATEGO faster. Section 12-A outlines potential work that can reduce the training time by optimization.

During the experiments, we also discovered that the drone sometimes had trouble finding the POIs in fig. 13. We believe this is because the drone is having trouble getting in the range of where it can see the POI. As presented earlier, the drone is only able to examine a POI if it is within 0.75 meters of it. The reason it is having trouble getting into this range, we assume is because of the strict move actions we have combined with the fact that the drone is not hovering in a static position. We

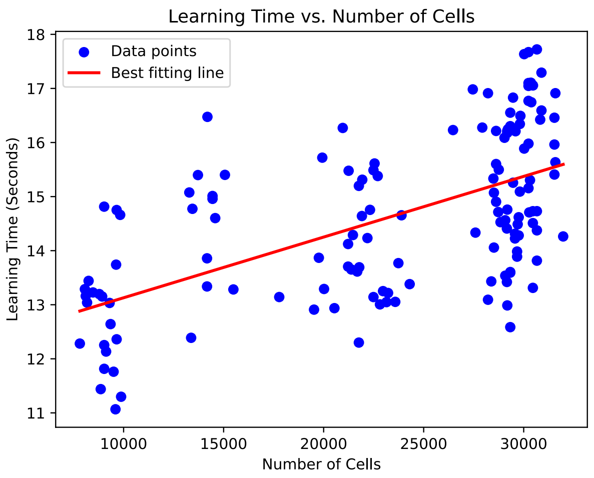

Fig. 15: Plot showing how the training time increases, as the number of cells in the map increases. The best-fitting line is calculated using linear regression.

observed that when a drone reaches this kind of limbo of being close to the POI, but never able to see it, it produces runs with a large number of times trained, actions activated and hitting the maximum time allowed for a run. This problem appeared in all experiments, both our approach and the baseline, and we believe this problem is reflected in the sometimes large standard deviation for the experiments. Possible solutions to this problem are presented in section 12-A.

C. Testing on different maps

To test the flexibility of the proposed solution, we test on a total of three different room setups, besides the one used for the previous experiments. These three rooms can be visualised in fig. 16, with room A used previously for the other experiments. The configuration of the solution that has been used for these experiments is UC-M1-H20 from table II. The rooms have a different number of POIs, various sizes and different layouts. For instance, room D in fig. 16 is a circular room, made for testing the capabilities of the solution in a room with curved walls. These experiments are not meant to find or show the best configuration setup for individual rooms, but merely that our proposed method can function in various environments. We present the results in the same way as the previous results, with suspected crashes being removed from the data. For each of the rooms, we collect 34 runs. It is important to note that the experiments for rooms B and D

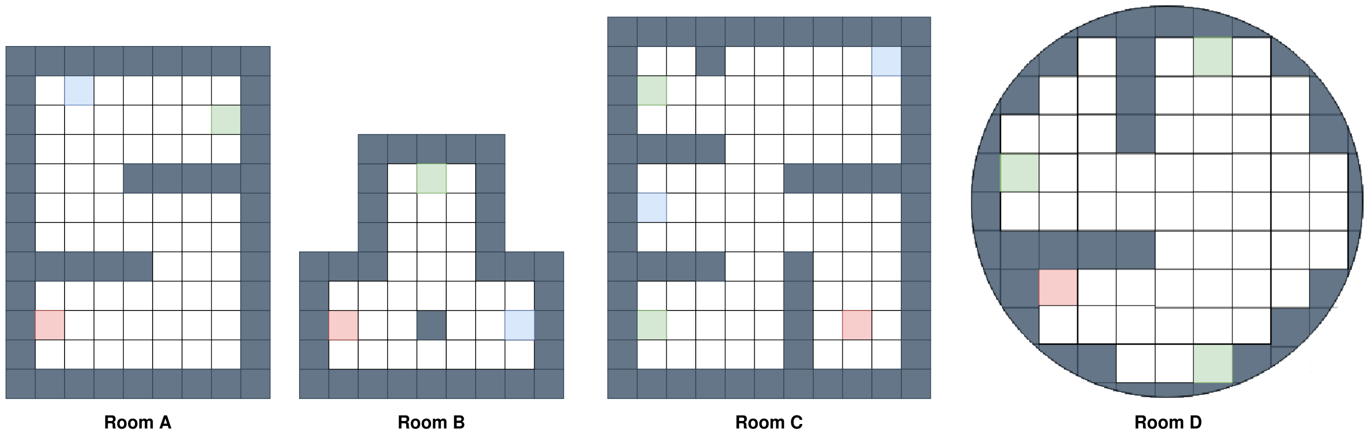

Fig. 16: The different rooms that we run experiments on. The scale of the rooms in this figure is incorrect. With a granularity of 0.05 room A has a total of 24.480 cells, room B has a total of 26.928 cells, room C has a total of 144.400 cells and room D has a total of 31.415 cells. The cell numbers are calculated as if the walls were not in the room. Room C is thus the largest room by a significant amount. The light blue cells are POI that are not pumps, while the green cells are POI that are pumps. The red cells are the location of the drone in its initial state.

both have the bug in UPPAAL STRATEGO presented earlier, while room C was done later with the bug fixed. The results can be seen in table IV.

We can see that the experiments for rooms B and D with the UC-M1-H20 configuration setup are showing similar results to the experiments for room A with the same setup. This is a sign that our proposed method is not limited to strictly square rooms but also functions reasonably well on other shapes. These three rooms are also roughly the same size. However, the data for room B shows that it had trouble fully exploring the map. This is also what led the experiment for room B to have a lower Completed Task compared to rooms A and D. In all runs it was able to examine all POIs, but in some cases, it was not able to fully explore it. We suspect this is because of the obstacle in the middle of the room causing both shields to block actions that would cause the drone to fully explore the room. We didn't notice any such drawbacks for room A or room D. The results from the experiment done in room C show our proposed method beginning to struggle with the large size of the room. Particularly we see completed tasks significantly lower than their counterparts, more crashes, longer completion times, it had to train more and a large number of actions activated. For this experiment, we also had to give the drone longer steps, meaning the drone can move 3 meters with one action instead of 1. The drone was still able to move half a meter. This was done as an attempt to not raise the horizon from 20, causing longer training time. The results from the room C experiment show that our method might begin to reach its limit when the rooms are getting very large. However, this might also show that the UC-M1-H20 configuration setup is not suited for rooms of this size, supporting our belief that having an *open* mapping configuration is the better choice for large open-spaced rooms. The number of crashes is also significantly greater in Room C. This may be because the drone can move up to 3 meters in this map, increasing the acceleration of the drone. Thus the safety constraint may need to be more strict, as the movement length increases.

D. Using the pipeline on a Turtlebot3

To fully prove the capabilities of the proposed pipeline, we attempt to use it on a Turtlebot3 [8] robot that can be seen in fig. 17. While the Turtlebot3 moves using wheels, it still shares many similarities with the X500 drone that we simulate in Gazebo. Just like the X500 drone, Turtlebot3 features both an odometry sensor and a LiDAR. However, the LiDAR on the Turtlebot has 360 vision, meaning that the Turtlebot can see everything around it. As mentioned in section 9-B, as long as the middleware publishes a LiDAR, Position and Controller topic, it fits into the pipeline, and the Turtlebot3 middleware does exactly that. The only thing that needs to be changed for the pipeline to work with the Turtlebot3, is how movement is handled in UPPAAL STRATEGO. Where a drone can move in certain directions without facing those directions, the Turtlebot3 can not. Therefore we have created a new UPPAAL STRATEGO model for the Turtlebot3, with the addition that the movement edges are only available if the robot is facing the way that the movement edge takes it.

The room that the Turtlebot3 will navigate in is a real-life creation of the model shown in fig. 13. For practical purposes, the model has been scaled down so that the width of the room is 240cm and the height is 360cm. The real-life room can be seen in fig. 18.

The experiment showed that the Turtlebot3 was successful at mapping the room, as well as locating the pump in the room. The experiment showed that the robot was able to map and find the POI in 3 minutes. While the experiment showed that the pipeline worked in real life, it did showcase several problems:

- **The odometry**: After driving for a short while the odometry data of the robot got very inaccurate, even

Setup	Completed Task	Suspected Crashes	Avg. Completion Time(minutes)	Avg. Number of Times Trained	Avg. training time(seconds)	Avg. Number of actions
Room B, 26.928 cells	87.10%	3	6.13 (±5.18)	13.29 (±13.07)	5.14 (±0.26)	113.45 (±116.91)
Room C, 144.400 cells	59.26%	7	15.32 (±3.17)	26.96 (±6.02)	10.34 (±0.24)	256.48 (±59.36)
Room D, 31.415 cells	96.77%	3	5.85 (±3.56)	11.03 (±7.57)	5.50 (±0.40)	96.03 (±65.82)

TABLE IV: The results from experiments with our proposed method on different rooms. The data is shown with the suspected crashes removed, meaning if there are 10 suspected crashes, and out of the 90 runs left, there were 45 completed, the results would be 50%. The **avg. completion time** is shown in minutes, with **avg. training time** in seconds. The number in the parentheses is the standard deviation.

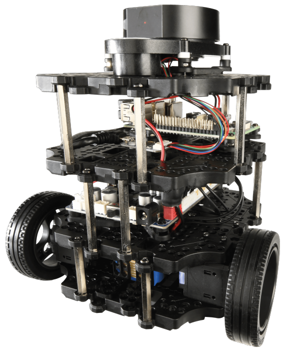

Fig. 17: The Turtlebot3 robot [8].

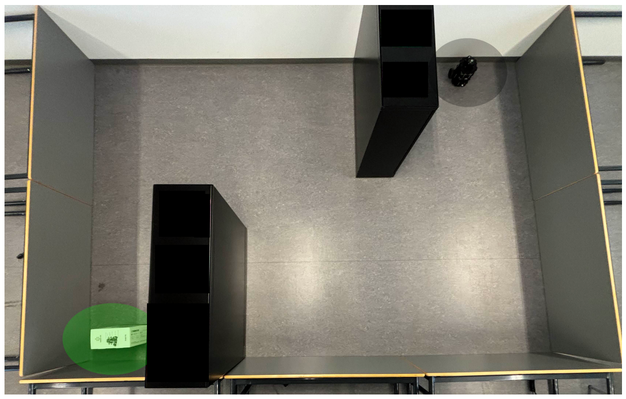

Fig. 18: The real-life room that Turtlebot3 will navigate and map in. The box in the green circle is the one and only POI in this room, and the Turtlebot3 is in the black circle.

while using SLAM. This resulted in the robot not driving straight. This drift in the odometry data, should not be this significant, and should not have happened this fast. A reason for this drift could be that the wheels on the Turtlebot3 were dirty when the experiment was conducted.

- **The LiDAR sensor**: When conducting the experiment, we found that the LiDAR sensor could not detect all kinds of materials. Particularly it had a hard time detecting the trash cans seen in the video (which is why there are placed another kind of material next to the trash cans). This may be due to the laser being reflected off the material. It is therefore important that the LiDAR sensor on the drone, that Grundfos will be using is of sufficient quality.

12. Conclusion & Future Works

As drones are increasingly being used to automate tasks, Grundfos seeks to find a way of using drones to automate pump inspections. In this paper, we have proposed a solution on how to automate these drone inspections. We propose a pipeline consisting of Uppaal Stratego to compute a reinforcement learning strategy, STOMPC as a controller, ROS to send and receive sensor data, Slam Toolbox to build a map of the environment and middleware to control the drone.

Throughout the experiments, we show that this pipeline works both in a simulated environment using Gazebo and in a real physical environment using a Turtlebot3 robot. Furthermore, we also show how the pipeline works in different environments such as large-scale rooms and rooms with curved walls.

All of the results from the experiments have been compared to a baseline algorithm, that uses Breadth-First-Search to compute a path for the drone. The first set of experiments showed that the baseline algorithm performed better on almost all recorded parameters, and on the ones where it didn't, it was marginally worse. These findings led us to discover bugs in the Uppaal Stratego model, as well as ideas on how to improve the reward function. The results after the improvements showed that Uppaal Stratego was now significantly better in some of the recorded parameters. For instance, using Uppaal Stratego the drone could on average reach the goal state using 33 fewer actions compared to using the baseline algorithm. In addition to these findings, we found that if one removed the training time, Uppaal Stratego was more than 1 minute faster than the baseline algorithm. These findings show the potential of Uppaal Stratego, if one manages to improve the learning time.

These findings illustrate the potential of Uppaal Stratego, provided that the learning time can be further optimized. Overall, this study highlights the feasibility and effectiveness of using reinforcement learning in collaboration with STOMPC, ROS and Slam Toolbox to automate drone inspections. The findings found in this paper can contribute to more efficient and reliable automated inspection systems in the future.

A. *Future Works*

With the pipeline being fully made, it makes sense to investigate how it can be optimized. For instance, one might look into how the map represented by a matrix in Uppaal Stratego can be optimized. One way is to partition the map, so that the parts of it that have already been fully explored are not a part of the map being trained on. Another way could be to make the matrix in UPPAAL STRATEGO a constant so it is not copied between states. Another idea is to ”shrink” the map, meaning that the map in UPPAAL STRATEGO has a higher granularity than the actual map. The granularity of 0.05 was picked so that the pumps could be detected as patterns in the map. However, it might not be necessary to have such a small granularity in the map in UPPAAL STRATEGO, if we can properly ”shrink” the map without losing obstacles and POIs. Other optimizations might be found by conducting experiments on UPPAAL STRATEGO learning parameters, reward engineering, and general code optimizations.

An addition that could reduce the impact of learning time, which has already been coded, but not tested, is beginning training before the control sequence has finished. The idea is that when there are a certain number of actions left in the control sequence, we predict where the drone will be after taking the remaining actions, and send that state to UPPAAL STRATEGO. Once the remaining actions have been taken, UPPAAL STRATEGO will already have calculated a new control sequence, or at least be close to it.

Another problem that must be investigated, is the problem of the drone not being able to get close enough to the pumps. To detect the pump, the drone must be within 0.75 meters of it, and it can only move 0.5 or 1 meter at a time. Meanwhile, it must also satisfy its safety constraint, which can be hard since pumps are often close to other objects. To reduce this problem, we can introduce a larger set of movement actions, such as being able to move 0.25 meters, or we can introduce a dynamic set of actions. For example, instead of telling the drone to move 1 meter north, UPPAAL STRATEGO should be able to tell the drone to move to a certain point in the map.

REFERENCES

[1] A. Rejeb, A. Abdollahi, K. Rejeb, and H. Treiblmaier, “Drones in agriculture: A review and bibliometric analysis,” *Computers and Electronics in Agriculture*, vol. 198, p. 107017, 2022.

[2] H.-W. Choi, H.-J. Kim, S.-K. Kim, and W. S. Na, “An overview of drone applications in the construction industry,” *Drones*, vol. 7, no. 8, 2023.

[3] M. Jaeger, P. Jensen, K. Larsen, A. Legay, S. Sedwards, and J. Taankvist, “Teaching stratego to play ball: Optimal synthesis for continuous space mdps,” in *Automated Technology for Verification and Analysis- 17th International Symposium, AVTA 2019, Proceedings* (Y.-F. Chen, C.-H. Cheng, and J. Esparza, eds.), Lecture Notes in Computer Science, (Germany), pp. 81–97, Springer, Oct. 2019. International Symposium on Automated Technology for Verification and Analysis, ATVA ; Conference date: 28-10-2019 Through 31-10-2019.

[4] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” 11 2022.

[5] S. Macenski and I. Jambrecic, “Slam toolbox: Slam for the dynamic world,” *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, 2021.

[6] O. Robotics, “Gazebo.” https://gazebosim.org/home. [Online; accessed 29-May-2024].

[7] N. Koenig and A. Howard, “Design and use paradigms for gazebo, an open-source multi-robot simulator,” in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, pp. 2149–2154 vol.3, 2004.

[8] O. Robotics, “TurtleBot3.” https://www.turtlebot.com/turtlebot3/. [Online; accessed 29-May-2024].

[9] M. Goorden, P. Jensen, K. Larsen, M. Samusev, J. Srba, and G. Zhao, “Stompc: Stochastic model-predictive control with uppaal stratego,” in *International Symposium on Automated Technology for Verification and Analysis* (A. Bouajjani, L. Holík, and Z. Wu, eds.), Lecture Notes in Computer Science, (Germany), pp. 327–333, Springer, 2022. 20th International Symposium on Automated Technology for Verification and Analysis, ATVA 2022 ; Conference date: 25-10-2022 Through 28-10-2022.

[10] B. Yamauchi, “A frontier-based approach for autonomous exploration,” in *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA’97. ’Towards New Computational Principles for Robotics and Automation’*, pp. 146–151, 1997.

[11] H. Li, Q. Zhang, and D. Zhao, “Deep reinforcement learning-based automatic exploration for navigation in unknown environment,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, pp. 1–13, 08 2019.

[12] F. Chen, S. Bai, T. Shan, and B. Englot, “Self-learning exploration and mapping for mobile robots via deep reinforcement learning,” 01 2019.

[13] S.-Y. Chen, Q.-F. He, and C.-F. Lai, “Deep reinforcement learning-based robot exploration for constructing map of unknown environment,” *Information Systems Frontiers*, vol. 26, 11 2021.

[14] S. Bøgh, P. Gjøl Jensen, M. Kristjansen, K. Guldstrand Larsen, and U. Nyman, “Distributed fleet management in noisy environments via model-predictive control,” *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 32, pp. 565–573, Jun. 2022.

[15] I. R. Hasrat, P. G. Jensen, K. G. Larsen, and J. Srba, “A toolchain for domestic heat-pump control using uppaal stratego,” *Science of Computer Programming*, vol. 230, p. 102987, 2023.

[16] H. Durrant-Whyte and T. Bailey, “Simultaneous localization and mapping: part i,” *IEEE Robotics & Automation Magazine*, vol. 13, no. 2, pp. 99–110, 2006.

[17] W. Hess, D. Kohler, H. Rapp, and D. Andor, “Real-time loop closure in 2d lidar slam,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1271–1278, 2016.

[18] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[19] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. USA: John Wiley & Sons, Inc., 1st ed., 1994.

[20] M. K. Axelsen, T. G. S. Lauritsen, K. L. Pedersen, and P. C. Greve, “Reinforcement Learning Through Uppaal in Simulation Environments: Pump Identification by Drones in Technical Rooms.” https://kbdk-aub.primo.exlibrisgroup.com/, 2024.

[21] Holybro, “PX4 Development Kit - X500 v2.” https://holybro.com/products/px4-development-kit-x500-v2. [Online; accessed 29-May-2024].

[22] K. G. Larsen, M. Mikučionis, and J. H. Taankvist, *Safe and Optimal Adaptive Cruise Control*, pp. 260–277. Cham: Springer International Publishing, 2015.

[23] GYM, “Frozen lake.” https://www.gymlibrary.dev/environments/toy_text/frozen_lake/. [Online; accessed 05-June-2024].

[24] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, “Safe reinforcement learning via shielding,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, Apr. 2018.

[25] T. A. N. Heirung, J. Paulson, J. O’Leary, and A. Mesbah, “Stochastic model predictive control — how does it work?,” *Computers & Chemical Engineering*, vol. 114, 11 2017.

[26] PX4, “PX4 homepage.” https://px4.io/. [Online; accessed 29-May-2024].

[27] H. V. Abeywickrama, B. A. Jayawickrama, Y. He, and E. Dutkiewicz, “Empirical power consumption model for uavs,” in *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, pp. 1–5, 2018.

APPENDIX

APPENDIX A
UPPAAL STRATEGO MOVING TEMPLATE

The UPPAAL STRATEGO Moving Template can be seen in fig. 19.

APPENDIX B
UPPAAL STRATEGO TURNING TEMPLATE

The UPPAAL STRATEGO Turning Template can be seen in fig. 20.

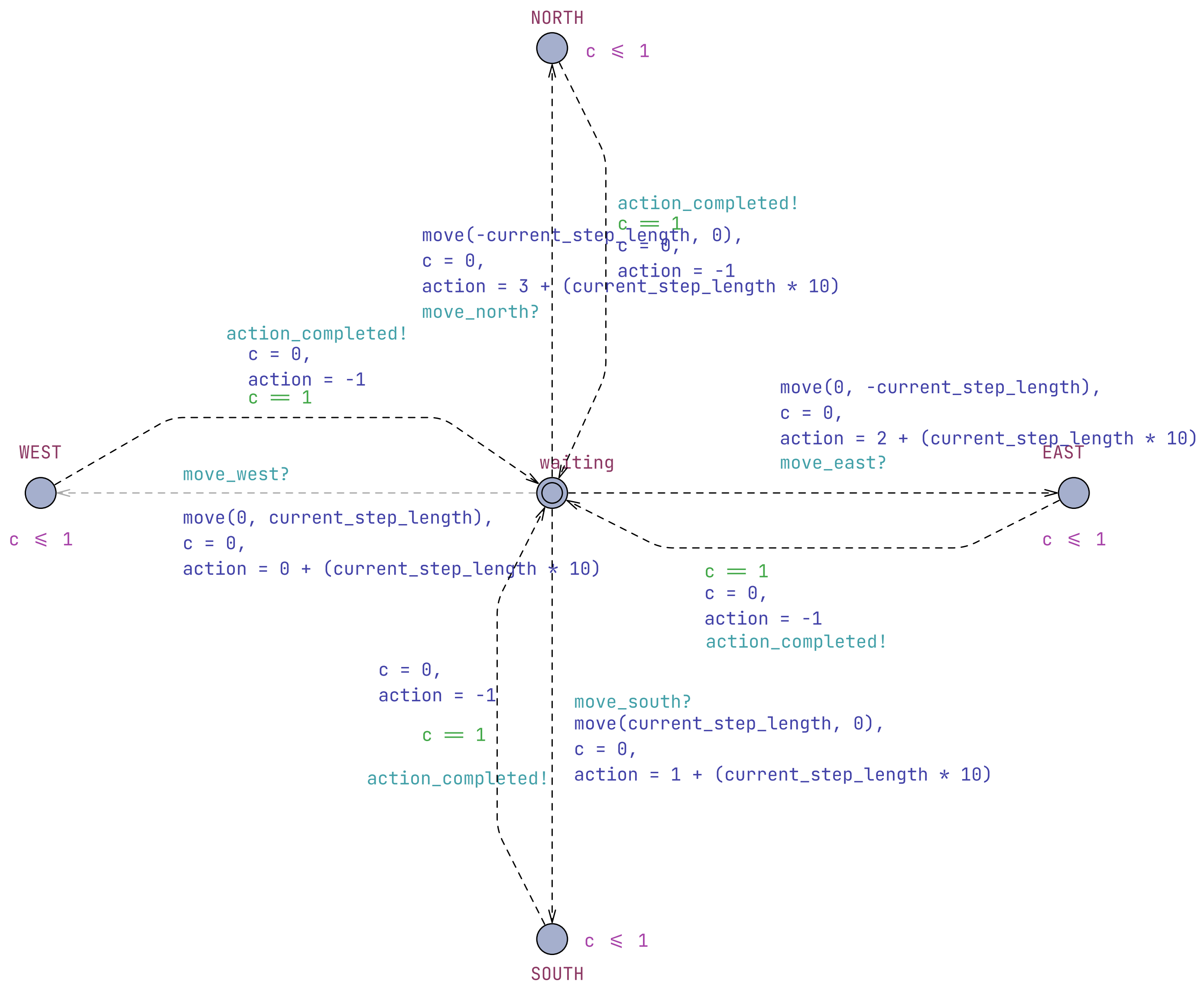
NORTH
c ⩽ 1
action_completed!
c == 1
c = 0,
action = -1
move(-current_step_length, 0),
c = 0,
action = 3 + (current_step_length * 10)
move_north?
action_completed!
c = 0,
action = -1
c == 1
move(0, -current_step_length),
c = 0,
action = 2 + (current_step_length * 10)
move_east?
WEST
move_west?
waiting
EAST
c ⩽ 1
move(0, current_step_length),
c = 0,
action = 0 + (current_step_length * 10)
c ⩽ 1
c == 1
c = 0,
action = -1
action_completed!
c = 0,
action = -1
c == 1
action_completed!
move_south?
move(current_step_length, 0),
c = 0,
action = 1 + (current_step_length * 10)
c ⩽ 1
SOUTH

Fig. 19: UPPAAL STRATEGO Moving Actions Template

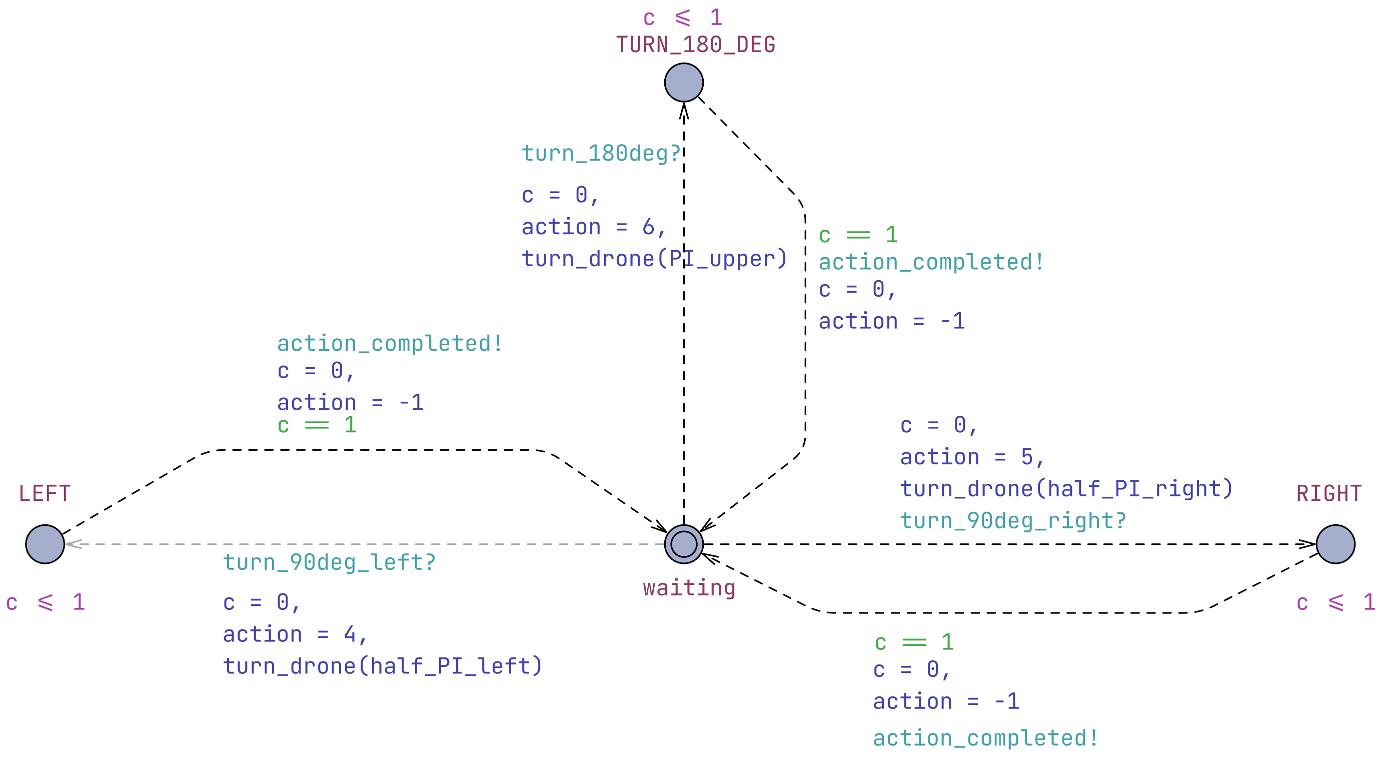

Fig. 20: UPPAAL STRATEGO Turning Actions Template