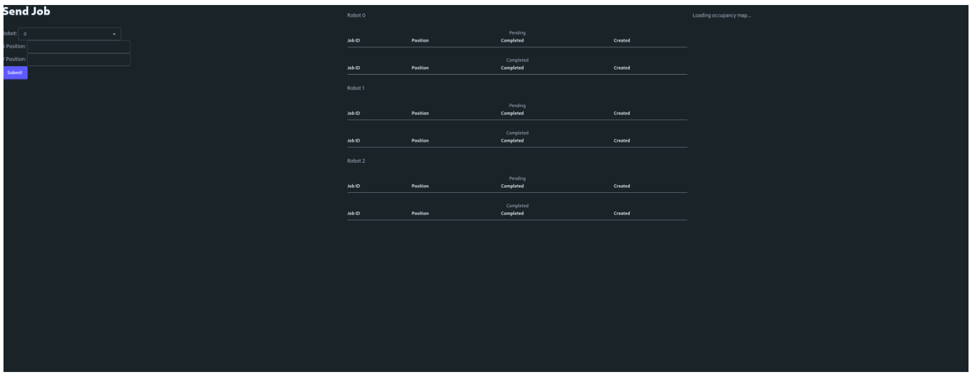
Software engineering for cyber-physical systems

For this project, Python was used to write the code to control the robots that run in Webots. For the dashboard, the SvelteKit framework was used with TypeScript.



Send Job

Robot: 0

X Position:

Y Position:

Submit

Robot 0

Pending

Job ID	Position	Completed	Created

Completed

Job ID	Position	Completed	Created

Robot 1

Pending

Job ID	Position	Completed	Created

Completed

Job ID	Position	Completed	Created

Robot 2

Pending

Job ID	Position	Completed	Created

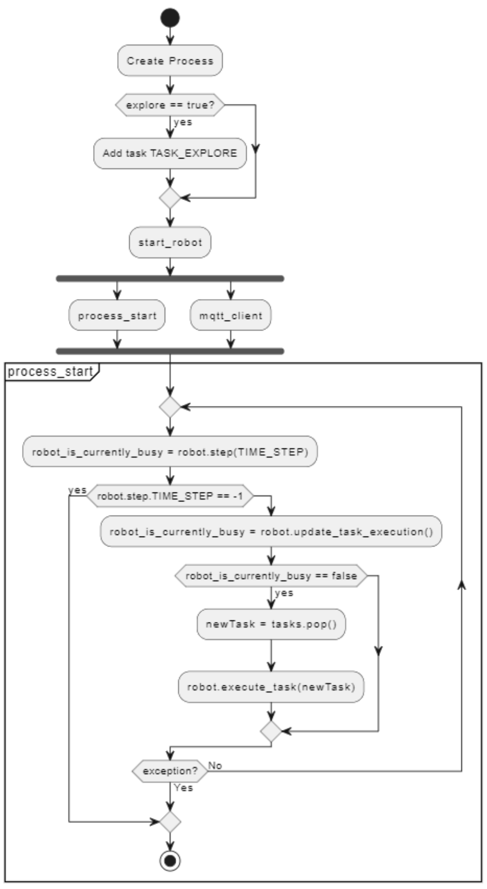
Completed

Job ID	Position	Completed	Created

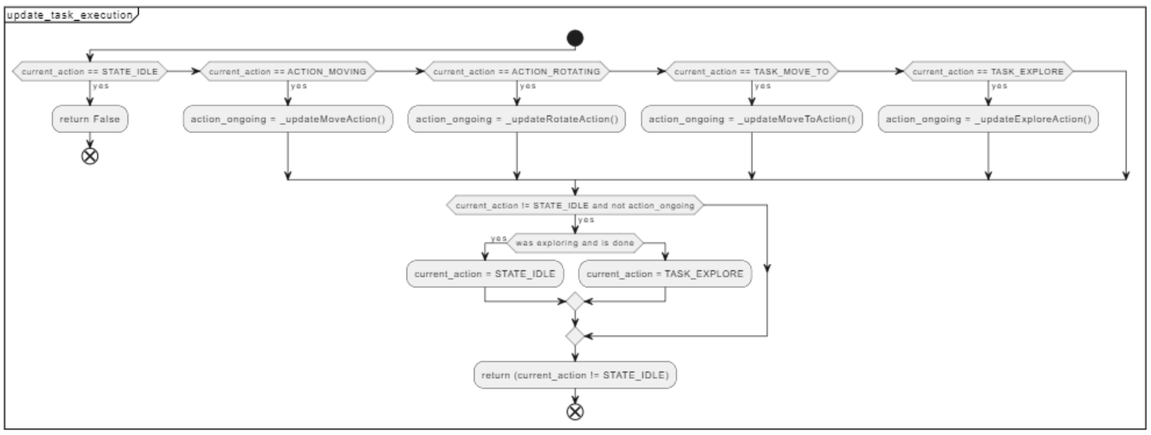
Loading occupancy map...

State Machine:

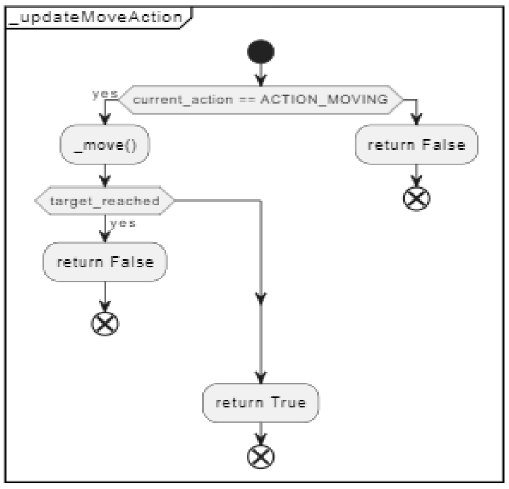
The Plantuml state machine models a robotic process that can handle various tasks that are needed to fulfil the client's wishes that the robots need to be able to do. It begins by creating a process and, if exploration is enabled, adds an exploration task before starting the robot and giving it two different functions: process_start and mqtt_client. The process_start loops repeatedly to check if the robot is busy or available. Task execution is delegated to specific functions based on the task type, each returning whether to stop or continue.

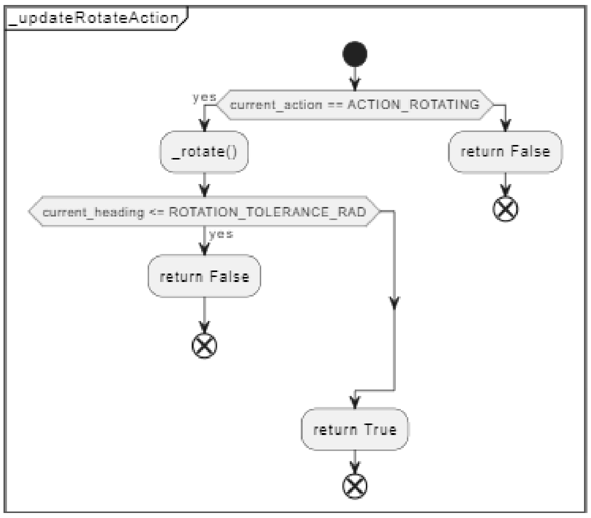

Create Process
explore == true?
yes
Add task TASK_EXPLORE
start_robot
process_start
mqtt_client
process_start
robot_is_currently_busy = robot.step(TIME_STEP)
yes
robot.step.TIME_STEP == -1
robot_is_currently_busy = robot.update_task_execution()
robot_is_currently_busy == false
yes
newTask = tasks.pop()
robot.execute_task(newTask)
exception?
No
Yes

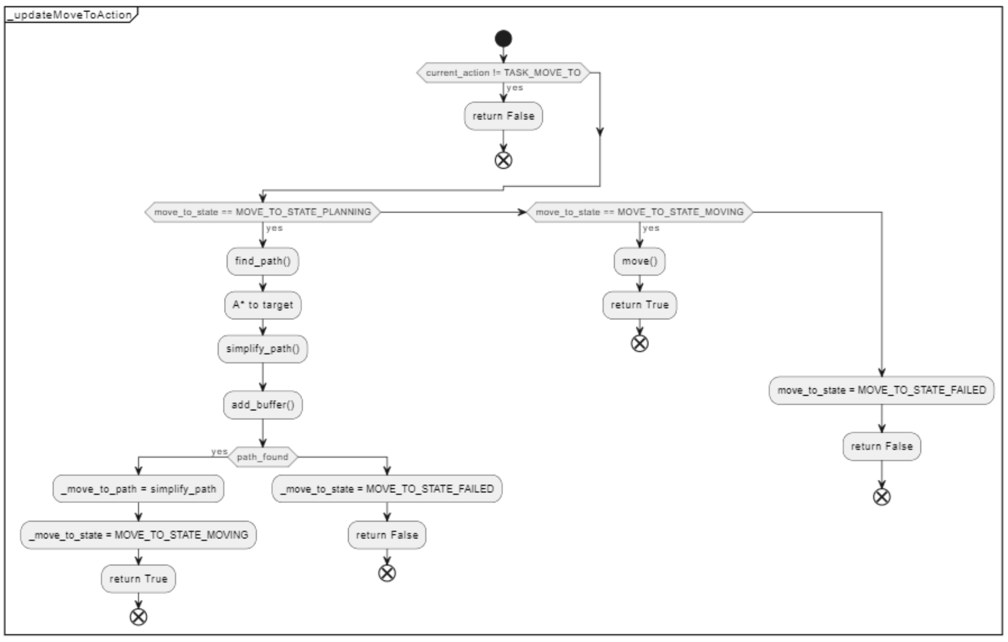
The update_taks_exectuiton component checks the current action and routes it to an update function depending on what the robot is doing at that given moment. These functions are used internally to determine the state and provide feedback to assess progress.


update_task_execution
current_action == STATE_IDLE
yes
return False
current_action == ACTION_MOVING
yes
action_ongoing = _updateMoveAction()
current_action == ACTION_ROTATING
yes
action_ongoing = _updateRotateAction()
current_action == TASK_MOVE_TO
yes
action_ongoing = _updateMoveToAction()
current_action == TASK_EXPLORE
yes
action_ongoing = _updateExploreAction()
current_action != STATE_IDLE and not action_ongoing
yes
was exploring and is done
yes
current_action = STATE_IDLE
current_action = TASK_EXPLORE
return (current_action != STATE_IDLE)

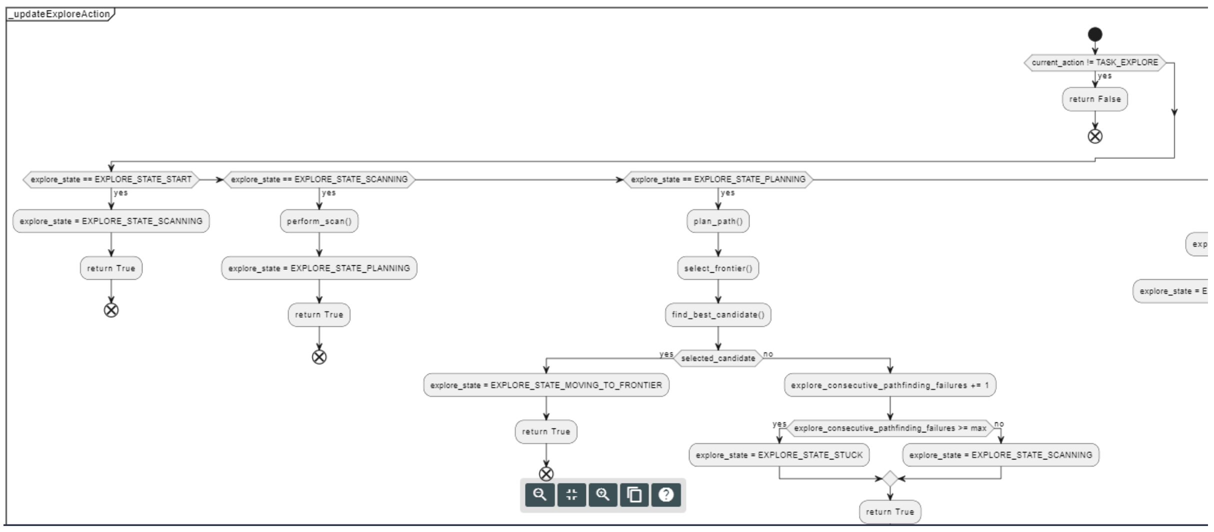
The updateMoveAction and updateRotateAction handle simple movement and rotation and checking whether the goal has been achieved. The updateMoveToAction performs a path planning (using the A* algorithm)

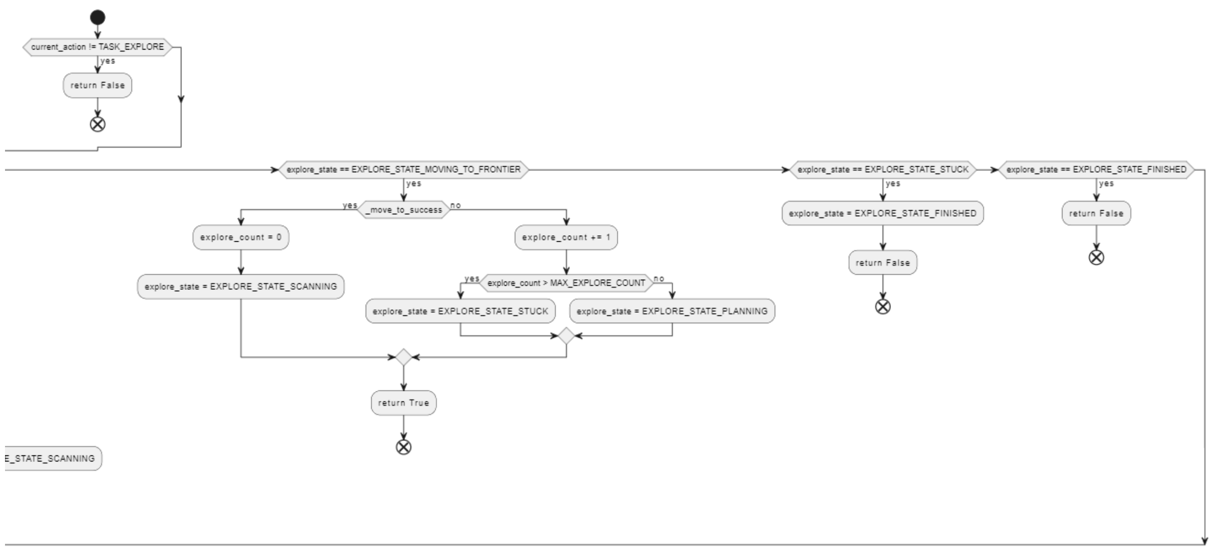



_updateRotateAction
current_action == ACTION_ROTATING
yes
_rotate()
return False
current_heading <= ROTATION_TOLERANCE_RAD
yes
return False
return True

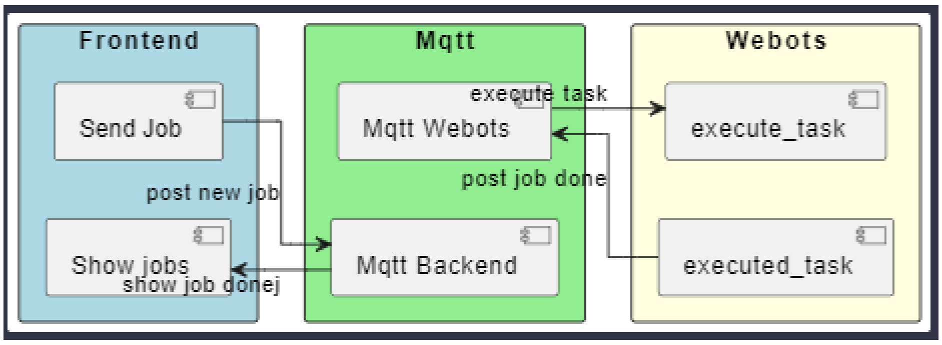

_updateMoveToAction
current_action != TASK_MOVE_TO
yes
return False
move_to_state == MOVE_TO_STATE_PLANNING
yes
find_path()
A* to target
simplify_path()
add_buffer()
path_found
yes
_move_to_path = simplify_path
_move_to_state = MOVE_TO_STATE_MOVING
return True
_move_to_state = MOVE_TO_STATE_FAILED
return False
move_to_state == MOVE_TO_STATE_MOVING
yes
move()
return True
move_to_state = MOVE_TO_STATE_FAILED
return False

The exploration routine is defined in updateExploreAction, which includes scanning, path planning, moving to a frontier, and handling failure conditions. If the robot gets stuck or finishes exploring, this function will update the internal state accordingly.


_updateExploreAction
current_action != TASK_EXPLORE
yes
return False
explore_state == EXPLORE_STATE_START
yes
explore_state = EXPLORE_STATE_SCANNING
return True
explore_state == EXPLORE_STATE_SCANNING
yes
perform_scan()
explore_state = EXPLORE_STATE_PLANNING
return True
explore_state == EXPLORE_STATE_PLANNING
yes
plan_path()
select_frontier()
find_best_candidate()
yes
selected_candidate
no
explore_state = EXPLORE_STATE_MOVING_TO_FRONTIER
return True
explore_consecutive_pathfinding_failures += 1
yes
explore_consecutive_pathfinding_failures >= max
no
explore_state = EXPLORE_STATE_STUCK
explore_state = EXPLORE_STATE_SCANNING
return True
exp
explore_state = E


current_action != TASK_EXPLORE
yes
return False
explore_state == EXPLORE_STATE_MOVING_TO_FRONTIER
yes
yes
_move_to_success
no
explore_count = 0
explore_state = EXPLORE_STATE_SCANNING
explore_count += 1
yes
explore_count > MAX_EXPLORE_COUNT
no
explore_state = EXPLORE_STATE_STUCK
explore_state = EXPLORE_STATE_PLANNING
return True
E_STATE_SCANNING
explore_state == EXPLORE_STATE_STUCK
yes
explore_state = EXPLORE_STATE_FINISHED
return False
explore_state == EXPLORE_STATE_FINISHED
yes
return False

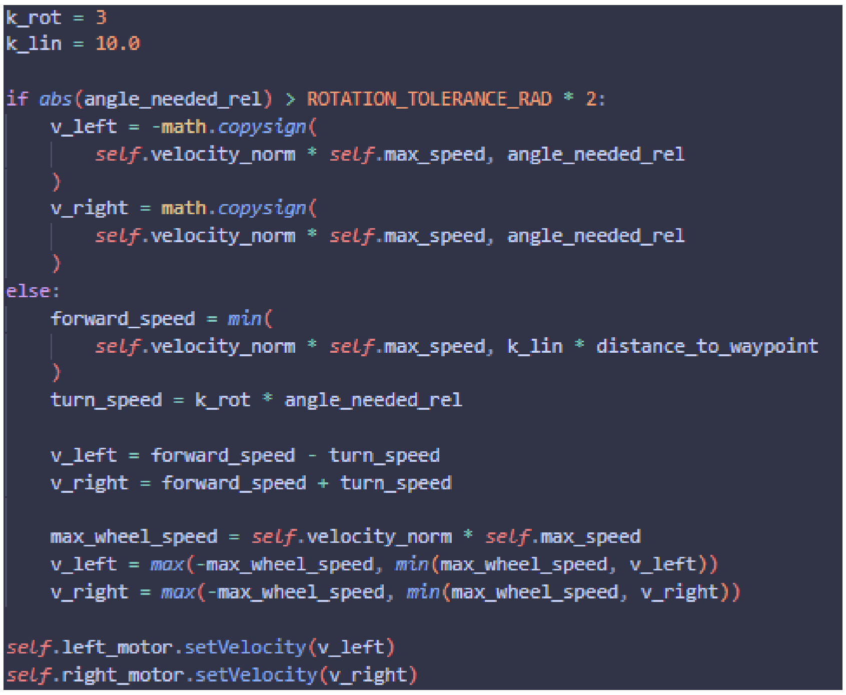
MQTT



This diagram illustrates a system architecture for job execution using MQTT messaging between the frontend, the MQTT backend and the Webots simulation environment. The user can submit jobs on the front end and view their status. The MQTT backend will receive and communicate those job requests to the MQTT Webots. These will send the execute_taks command to the webots itself. When the task is completed, the Webots will send the executed_task to the MQTT Webots, translating it to "post job done”. This message will allow the front end to update and display the job completion status.

Movement of the differential-drive robot

This code controls the movement of the robot. It uses two constant values: k_rot for rotation and k_lin for forward movement sensitivity. If the robot wants to turn far from the target, it sets the wheel speed on opposite sides at opposite speeds. Otherwise, it calculates a forward speed based on the distance to the waypoint and a turning speed based on the angle difference. These speeds are then combined for the left and right wheel velocities. The turning is achieved by subtracting or adding the turn_speed from the forward_speed. A maximum speed is guaranteed to ensure the robot will remain safe. When the speed is calculated for each wheel, the motors will simultaneously be turned up to the correct speed.



```python
k_rot = 3
k_lin = 10.0

if abs(angle_needed_rel) > ROTATION_TOLERANCE_RAD * 2:
    v_left = -math.copysign(
        self.velocity_norm * self.max_speed, angle_needed_rel
    )
    v_right = math.copysign(
        self.velocity_norm * self.max_speed, angle_needed_rel
    )
else:
    forward_speed = min(
        self.velocity_norm * self.max_speed, k_lin * distance_to_waypoint
    )
    turn_speed = k_rot * angle_needed_rel

    v_left = forward_speed - turn_speed
    v_right = forward_speed + turn_speed

    max_wheel_speed = self.velocity_norm * self.max_speed
    v_left = max(-max_wheel_speed, min(max_wheel_speed, v_left))
    v_right = max(-max_wheel_speed, min(max_wheel_speed, v_right))

self.left_motor.setVelocity(v_left)
self.right_motor.setVelocity(v_right)
```