Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Jonas Fillan

# Autonomous inspection and maintenance missions with AI planning and the ROSPlan framework

Master's thesis in Cybernetics and Robotics
Supervisor: Aksel A. Transeth (SINTEF), Anastasios Lekkas (NTNU)
Co-supervisor: Synne Fossøy (SINTEF), Maria-Efstathia Tsiourva (SINTEF)
May 2023

NTNU
Norwegian University of Science and Technology

Jonas Fillan

# Autonomous inspection and maintenance missions with AI planning and the ROSPlan framework

Master's thesis in Cybernetics and Robotics
Supervisor: Aksel A. Transeth (SINTEF), Anastasios Lekkas (NTNU)
Co-supervisor: Synne Fossøy (SINTEF), Maria-Efstathia Tsiourva (SINTEF)
May 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

# Abstract

The demand for automated solutions is ever-growing in almost all aspects of the modern world. The industrial sector has always been at the forefront of automation and in the days of advanced computation and Artificial Intelligence (AI), it is pushing to automate even more tasks that have traditionally been handled by humans. Advancements in Guidance, Navigation, and Control (GNC), perception, and sensor fusion has made autonomous robotics a feasible reality for automating industrial tasks. However, important capabilities of higher level planning and reasoning are fields of study that has not received enough attention in order to be deployable on a larger scale.

Automated planning and acting (AI planning) represents a methodology within the field of AI that enables higher level mission planning. AI-planning has been a field of study for decades, but due to constraints with regard to robotic capabilities, computational power, situational awareness, and robustness, it has to a large degree been of academic interest and for very special use-cases. Through the research project ROBPLAN, that this thesis is a part of, SINTEF aims to develop methods for AI planning, using mobile manipulators and unmanned aerial vehicles (UAVs) for Inspection and Maintenance (I&M) missions. The aim of this thesis is to contribute to the ROBPLAN project by answering the research question: "*How can AI Planning aid in autonomous robotic inspection and maintenance missions in industrial environments?*"

Using the ROSPlan framework, and extending its functionality with additional supporting features, this thesis has implemented and tested a robotic system that is capable of performing I&M missions in an industrial environment. The thesis has proposed solutions for common problems related to the dispatch of plans generated using AI planning, such as handling unplanned events due to a dynamic environment, re-planning capabilities, and operator-in-the-loop systems. A Planning Domain Definition Language (PDDL) domain and problem design for solving autonomous robotic I&M missions was also proposed.

All systems and designs proposed in this thesis has been implemented and tested, both in simulation and small scale lab experiments. The results of the test showed that the systems implemented helped increase the robustness of an AI planning assisted robotic mission by making it able to handle unplanned events in a dynamic mission environment better than without such systems. However, for the system to be deployable in on a larger scale, further development is needed to ensure robustness and safety. The results of this thesis show that AI planning can indeed aid autonomous robotic inspection and maintenance missions by adding high level planning capabilities to the system. However, the AI planner in itself is not enough to solve this task, and extensive supporting infrastructure, like the proposed solutions in this thesis, is needed in order to contribute to robustness during mission dispatch.

# Preface

This thesis is the final delivery of the subject *TTK4900 - Engineering Cybernetics, Master's Thesis*, and marks the end of two years of studies at the master's degree program for Cybernetics and Robotics at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway. The thesis is a continuation of the pre-project from fall 2022, "PDDL plan validation system using ROSPlan and a TurtleBot3 simulator", and was written for SINTEF Digital, Department of Mathematics and Cybernetics.

I would like to thank SINTEF Digital for facilitating this project, with a special thanks to my supervisor Aksel Andreas Transeth for all the help and support throughout the project. A big thanks go out to my co-supervisors Synne Fossøy and Maria-Efstathia Tsiourva at SINTEF for support throughout the master's project and pre-project respectively. I also owe thanks to my NTNU supervisor Anastasios Lekkas for lending equipment for the lab testing, and to Stefano Brevik Bertelli for providing test locations at very short notice.

This project has been long and demanding, and without the help and moral support from my group of friends at the office, the semester would not have been the same. Thank you for all the coffee breaks, waffle Fridays, technical talks, and procrastination opportunities.

Last, but definitely not least, I would like to thank my girlfriend, Ia, for all the moral support to help me through this project.

Jonas Fillan 01.06.23

Jonas Fillan Date

# Table of Contents

# Acronyms

**AI** Artificial Intelligence.

**AI planning** Automated planning and acting.

**GNC** Guidance, Navigation, and Control.

**GUI** Graphical User Interface.

**I&M** Inspection and Maintenance.

**IDL** Interface Definition Language.

**IMU** Inertial Measurement Unit.

**IPC** International Planning Competition.

**IR** Infrared.

**KCL** King's College London.

**LPG** Local search for Planning Graphs.

**MCU** Microcontroller Unit.

**PDDL** Planning Domain Definition Language.

**POPF** Partial Order Planning Forwards.

**ROS** Robot Operating System.

**SBC** Single Board Computer.

**SDK** Software Development Kit.

**SLAM** Simultaneous Localization And Mapping.

**STRIPS** STanford Research Institute Problem Solver.

# 1

# Introduction

## 1.1 Motivation

The continued growth in demand for automated and autonomous robotic solutions within the industrial sector has resulted in a demand for enhanced mission planning capabilities for autonomous robots [1]. Automating tasks that traditionally have been handled by humans can be an effective measure in cutting cost, increasing productivity, and reducing health and safety risks. An example of tasks that are subject to automation using autonomous robots is Inspection and Maintenance (I&M) missions. I&M missions can include tasks like photographing areas of interest, monitoring specific areas, mapping areas using sensors, and manipulating object in the environment using robotic manipulators. Industrial environments are often very complex and operating robotic platforms in these environments is often challenging and unpredictable, necessitating the need for adaptive mission planning to enable robust and predictable execution of autonomous missions. This may involve consideration of unexpected obstacles, changes in the environment, emergency situations, or human interaction.

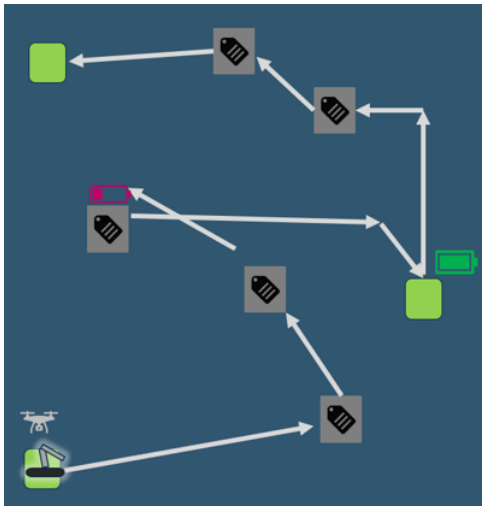


**Figure 1.1:** Illustration of an example for a robotic inspection mission. This illustration shows the inspection of 5 car seals, as well as the robot taking battery conditions into account. The illustration is reused, with permission from SINTEF, from internal documents of the SINTEF ROBPLAN project [1]

Automated planning and acting (AI planning) represents a methodology for mission planning that enables an agent to plan a mission based on the abilities and constraints of the agent and the environment it operates in. AI-planning has been a field of study for decades, starting with the STanford Research Institute Problem Solver (STRIPS) in the 1970s [2], and has been covered extensively. However, due to constraints with regard to robotic capabilities, computational

power, situational awareness, and robustness, it has to a large degree been of academic interest and for very special use cases, like NASA's Mars Exploration Rovers [3]. This is slowly changing as technologies within these fields are advancing, and interest for AI-planning is now spreading within commercial industries. Through the ROBPLAN research project, SINTEF is focusing on the development of AI planning techniques and demonstrating their applications in the context of inspection and maintenance missions, using mobile manipulators and unmanned aerial vehicles (UAVs) as case studies [1]. These missions include tasks like inspection of pipelines, monitoring gauges and indicators, reporting irregularities, and verifying the presence of markers and valve car seals, markers used to verify that a valve has not been operated unauthorized. Figure 1.1 demonstrates an example of an envisioned robotic inspection mission at an industrial production facility. In this scenario, the robot must plan and execute an inspection mission involving 5 valve car seals. The system is capable of inspecting objects using both a normal and Infrared (IR) camera, as well as taking into account factors such as battery level and potential disruptions to its path.

This project aims to contribute to the ROBPLAN research project by investigating how AI planning can contribute to autonomous Inspection and Maintenance (I&M) missions, like the one described above, and what supporting systems are needed around the AI planner in order to enable robust mission dispatch. The project will utilize the Robot Operating System (ROS) and ROSPlan AI planning framework, and the functionalities of the ROSPlan framework will be expanded with features that are needed in order to perform reliable robotic missions. To test the performance of the functionalities developed in this thesis, the ROS simulation tool developed in the preparation project for this thesis [4] will be utilized, as well as a physical implementation on a robotic platform.

## 1.2 Problem description and contribution

The overall goal of this thesis is to extend the knowledge within the field of AI planning in robotics. The thesis aims to investigate how AI planning can be used together with an Inspection and Maintenance (I&M) robot in order to automate tasks that traditionally have been handled by humans. This includes how to set up the AI planning system, as well as supporting infrastructure around the AI planner to enable the generated plan to be performed.

The main research question for the project is:

"*How can AI Planning aid in autonomous robotic inspection and maintenance missions in industrial environments?*"

With this research question in mind, the project is broken down to three key tasks:

- Develop a Planning Domain Definition Language (PDDL) domain description that defines the capabilities of an inspection robot, and a problem instance that describes an I&M mission like the one in section1.1.
- Implement an AI planning system with ROSPlan to extend its functionality for I&M missions in industrial environments.
- Test and evaluate the extended functionality of the ROSPlan AI planning system through simulation and lab tests, focusing in particular on how the overall system can handle changes and unexpected situations that occur during a mission.

By developing a PDDL domain description and problem instance for I&M missions, the thesis contributes a suggestion of how the PDDL design can be made versatile, allowing different robots with different capabilities to be used, including important aspects like robot battery restrictions into the planning problem. Investigating, and implementing, tools and techniques for performing autonomous robotic I&M missions using AI planning, the thesis presents a system that is not only capable of planning missions, but also able to dispatch the plan, handling unplanned events and disturbances during dispatch. This is functionality that is important when implementing AI planning solutions on a robotic system to be able to carry out the generated plan. Testing and evaluating the features developed in this project is done in order to validate the functionality of the system, including its ability to be adapted from a simulated environment to a physical robot.

## 1.3 Scope and Delimitations

This project is the final delivery of the subject *TTK4900 - Master's Thesis*. This is a standalone project, but closely tied to the Specialization Project report (*TTK4551*) that served as a preliminary project for this thesis. *TTK4900* is a 30 SP subject, and corresponds to approximately 40 hours of work per week. The project was started 10th of January 2023, with a submission date of 6th of June 2023. The scope of the project will be outlined, but not necessarily limited by the curriculum of the candidate's field of study, as well as relevant theory from previous studies.

This thesis is part of the SINTEF ROBPLAN project [1] and the scope and delimitations of the thesis is guided by the project assignment given by SINTEF. The focus of this thesis is mainly on the supporting systems around the AI planner that allows the planner to operate and the generated plans to be performed by the robot. As the amount of research on planning algorithms and AI planners is massive and well established, the inner workings of AI planners is not a focus of this thesis and is only described where context is needed. As AI planning involves high level planning, lower level control like Guidance, Navigation, and Control (GNC) is outside the scope of this thesis, and will also only be mentioned when context is needed.

The systems developed in this project have been tested, both through simulation and lab experiments, with a small-scale robotic platform. However, due to time constraint, real world tests with a full scale I&M robot were not feasible.

Unless otherwise specified in the figure captions, all illustrations and diagrams used in this thesis were created by the author.

## 1.4 Structure

In chapter 2, relevant **background** information that is important for the thesis is presented. A presentation of the background and theory of AI planning is provided, as well as the background and details of the Planning Domain Definition Language (PDDL). A brief description of AI planners is provided, and details about the specific AI planners used in this project is also explained. To provide some context for the experiments and development work done in this project, the ROSPlan framework and its planning system is outlined. Section 2.6 provides a description of relevant literature within the field of AI planning that may be relevant for this thesis. Some parts of the background chapter are directly taken from the specialization project report [4] that served as a preparatory project for this thesis. For the parts that this applies to, it will be clearly stated at the start of the section.

Chapter 3 presents the PDDL domain description and problem instance that was designed for this project is an effort to answer the research questions. Chapter 4 gives insight into the work that has been put into setting up and extending the functionality of the ROSPlan framework. Also this chapter contains some material from the specialization project.

Chapter 5 describes the obtained **results** of the project. It also includes a **discussion** of the results, putting them into context in an effort to answer the research questions.

Finally, chapter 6 summarizes the results and discussion to form a **conclusion**. This chapter also describes suggestions for **further work**.

# 2

# Background

In this chapter, background information that is important for the rest of this thesis is presented. A brief introduction to AI planning, PDDL and AI planners is provided, as well as some insight into the tools used for the project. The chapter is caped off with an overview of literature and works that are relevant for the work presented in this thesis.

Due to this project reusing parts of the systems developed in the pre-project for this thesis, parts of this chapter are reused from the project report [4]. This applies specifically to sections 2.4 and 2.5 in their entirety, as well as the initial parts of section 2.2

## 2.1 Automated Planning and Acting (AI Planning)

The ability to perform tasks and missions independently in a dynamic environment, without relying heavily on human interaction, necessitates the use of *deliberative acting*. According to Ghallab, Nau, and Traverso [5], a deliberative acting *agent* is *"motivated by some intended objective"* and employs logical reasoning to rationalize executing particular actions. Russell and Norvig [6] uses the term *knowledge-based agents* and describes them as an agent using "*a process of reasoning over an internal representation of knowledge*". Although autonomous systems can perform tasks without deliberative acting, it is crucial when faced with changes and disruptions in a constantly changing environment.

AI planning is an AI approach that is grounded in the classical planning theory. Russell and Norvig [6] defines classical planning as the task of finding an actions sequence that can achieve a specific goal in a discrete, deterministic, static and fully observable domain. Classical planning has been a topic of continued study for decades, starting all the way back in the 1970s with the STanford Research Institute Problem Solver (STRIPS) [2]. Although research on planning algorithms continued throughout the 70s and 80s, the focus was mostly on simple puzzle problems like Blocks World and the Tower of Hanoi, and the lack of progress towards practical applications led to doubt of its utility within the *AI* community [7]. This started changing during the 90s, along with the development of problem-specific planners, making the solvable problems more realistic and ap-

plicable to real-life scenarios. This helped push classical planning research towards solving more realistic problems, leading planners capable of producing large scale plans with short processing times [7].

Around the turn of the century, a large driving force within the planning research field was created in the form of the International Planning Competition. To facilitate this competition, a standardized language called Planning Domain Definition Language (PDDL) [8] was developed. This modeling language made it possible to define large planning problems using a general domain description and a specific problem instance, and PDDL has later become the de facto standard description language for planning problems. A more detailed description of PDDL is provided in section 2.2.

In the literature on classical planning and AI planning, many use cases within different fields have been proposed. Examples of such use cases include logistics and supply chain management, production planning, space exploration, underwater robotics, and agriculture [3], [9], [10].

AI planning represents high-level control in a system. In regard to robotics, this means that when dividing the robotic control system into control levels based on the level of abstraction away from hardware, AI planning represents the highest and most abstract level, farthest away from the robotic hardware. Figure 2.1 illustrates an example of how a robotic system can be divided into levels. In this thesis, only the top level of the figure is part of the scope.

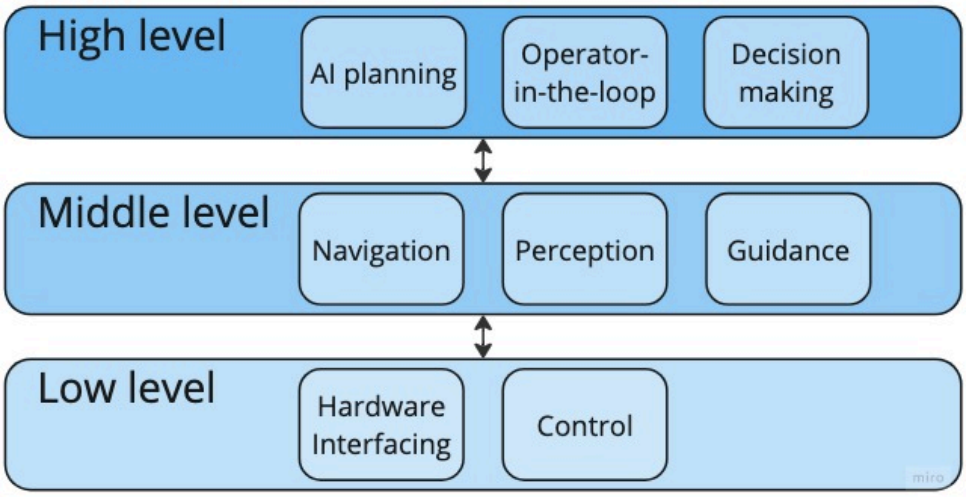



**Figure 2.1:** Illustration of how a robotic system can be divided, based on control levels. In this thesis, the focus is on the top level. Note that a system can be divided in different ways and will vary based on the specific system architecture.

## 2.2 Planning Domain Definition Language (PDDL)

The Planning Domain Definition Language (PDDL) [8] is a modeling language used to define the capabilities and constraints of a system, often called the *domain*. It is typically used in conjunction with a planning engine, which uses the information defined in the domain to automatically generate a plan of action to reach a pre-defined goal.

PDDL is the standard language for defining planning domains in the annual International Planning Competition (IPC) and was first introduced by Drew McDermott, et al. [8], for the 1998 AIPS Planning Competition. The language allows a planning problem to be defined as a domain description with predicates, numeric functions, and actions [8]. Since its introduction, PDDL has developed into several new evolutions, introducing more features and making the language more versatile and powerful. The newest version of PDDL that is used in the IPC is PDDL 3.1, which has been used since 2011. There are also other variants or extensions of PDDL, like PDDL+ [11], that is not used in IPC.

When using PDDL with a planner, two files are used. The domain description file defines all the universal and constant aspects of a planning problem and does not change as time progress. The other part of the PDDL planner input is the problem definition. This file describes the problem instance that the planner is solving and includes objects related to the problem, initial conditions and the goals that are to be reach through the plan [8]. The domain description can be said to be a general description of a planning domain, while the problem instance is the specific problem or missionthe planner is to solve. Many different problem instances can be run on the same domain.

### 2.2.1 PDDL domain

As mentioned above, the PDDL domain is a general representation of the planning domain. This generalization is achieved by using variables to define objects, predicates, and actions. Variables represent changeable entities, enabling actions to be applied to a diverse set of objects within the problem space. This broadens the action's applicability, thereby offering a scalable solution for large problems. Variables also create flexibility and abstraction by avoiding the need to specify every possible instance of an action or a state. By employing variables in this manner, PDDL domain definitions not only achieve a high level of reusability across different problems within the same domain, but also significantly enhance the flexibility of the planning system.

The domain description needs to follow a specific format in order for a PDDL compatible AI planner to be able to use it. This format is specified in the research paper for the introduction of the specific PDDL version. In this thesis, PDDL 2.1 is used [12] for all PDDL designs. A PDDL 2.1 domain description contains the following parts, and will be further explained below:

**Define (domain name),** defining the name of the specific domain.
**Requirements,** define what functionality requirements the PDDL planner must be able to handle.
**Types,** used to declare the types of objects that will be used in the domain.
**Predicates,** logical arguments that apply to specific types in the domain.
**Functions,** defines numeric quantities, like time or cost of an action.

**Actions,** Actions that can be performed by the system in order to move from one state to another. Needs a name, parameters, preconditions, and effects.

**Durative Actions,** similar to actions, but includes duration of action. Makes it possible to create temporal plans.

Code 2.1 illustrates an example of a PDDL domain for moving a robot between waypoints. From the first and second line, the file can be seen defined as a domain with the domain name *robot*. The third line consists of the *requirements* for the domain. In this domain, only typing and durative actions are included.

The next arguments in the domain file are *types* and *predicates*. In this example, waypoint and robot are the only entities that are used. Two predicates are defined for the domain. The *robot_at* predicate for a robot *v* is used to describe the location of the robot and will be true when it is located at a waypoint *wp*. *Visited* describes when the robot has been on a specific waypoint, and will be used to define mission goals in the problem instance. *Functions* defines numeric quantities and was implemented in PDDL version 2.1 [12]. In this example, a function is defined as the distance between two waypoints and will later be used to define the duration of a durative action

```
1
2  (define
3      (domain robot)
4      (:requirements :typing :durative-actions)
5      (:types
6    waypoint
7    robot
8      )
9      (:predicates
10   (robot_at ?v - robot ?wp - waypoint)
11   (visited ?wp - waypoint)
12     )
13
14     (:functions
15   (distance ?wp1 ?wp2 - waypoint)
16     )
17
18     ;; Move to any waypoint, avoiding terrain
19     (:durative-action goto_waypoint
20   :parameters (?v - robot ?from ?to - waypoint)
21   :duration ( = ?duration distance ?wp1 ?wp2)
22   :condition (and
23     (at start (robot_at ?v ?from))
24   :effect (and
25     (at end (visited ?to))
26     (at end (robot_at ?v ?to))
27     (at start (not (robot_at ?v ?from))))
28 )
```

**Code 2.1:** PDDL domain for moving a robot between waypoints

The last part of this example domain is the *action*. Several actions can be declared in a domain file. However, for this domain, there is only one action to reduce complexity. Here, the action in question is a durative action called *goto_waypoint* and is meant to command the robot to move to a specified waypoint. The parameters for the action are declared as a robot *v* and two waypoints,

*from* and *to*, that the robot is meant to move between. Duration is, as mentioned earlier, defined as the distance between the two waypoints. As numeric functions do not have a specific unit, duration does not have to be related to time. *Condition* and *effect* declare what the status before and after the action should be. The arguments are declared with a prefix that describes at what point in the action sequence the predicate should change. In this example, the *robot_at* predicate for the first waypoint will be removed at the start of the action, while the *robot_at* and *visited* predicate for the second waypoint will be set at the end of the action sequence. [12]

It is important to note that there are more arguments available for a PDDL domain than what is used in this simple domain example. A more comprehensive description of PDDL domains can be found in the PDDL design papers [8], [11], [12].

### 2.2.2 PDDL Problem

Just as the PDDL domain represents the general rules and dynamics of a planning environment, a PDDL problem instance provides a concrete scenario within that domain. It does so by specifying particular objects, their initial states, and the desired goal state. The problem instance enables the PDDL compatible AI planner to generate plans based on individual circumstances, while the domain provides the general rules applicable to all instances.

The problem instance leverages the abstraction in the domain by substituting the variables with specific objects or values, thereby giving a specific scenario to which the general actions, predicates, and requirements can be applied. This allows AI planners to generate concrete, actionable plans. The PDDL problem instance follows a well-defined structure, which, in the context of PDDL 2.1, includes the following parts:

**Define (problem name),** which defines the name of the specific problem instance.
**Domain,** specifies the domain within which the problem exists.
**Objects,** these are the specific entities that exist in this problem instance. They are instances of the types defined in the domain.
**Init,** defines the initial state of the problem, specifying the truth values of predicates involving the objects.
**Goal,** specifies the desired state or condition that a successful plan should achieve.
**Metric (optional),** used to define the optimization criterion for the plan, such as minimizing time or cost.

Code 2.2 is a short and simple example of a problem instance related to the domain in Code 2.1. In this example, four *objects* are defined, using the same types that were defined in the domain. Four initial conditions are declared in the problem, the first declaring the initial position of the robot and the next three defining the distance between different waypoints. Note that not all combinations of waypoints are declared in this example. PDDL operates under a *closed-world assumption*. I.e., any predicates that are not declared as true in the initial condition, will be assumed to be false. This should therefor mean that undeclared distances should not be included in the planning problem. However, some planners handle undeclared function values differently, and declaring all

values can be a good measure to avoid undesired effects in the plan. It is worth mentioned that the closed-world assumption can be omitted by declaring an *open-world* requirement in the domain [8], planner support for this is limited, however.

The last part of the problem file in code 2.2 is the *goal* description. A valid plan is a plan that is able to bring the system to a state where all goals are achieved without violating any constraints [8]. If the goals are not achievable from the initial conditions, the planner will fail to produce a valid plan. Lastly, in this example, a *metric* is not included to reduce complexity. Not defining a metric will make the planner find a plan without optimizing for any specific cost, just find a valid plan. In this example, a natural metric could be minimizing time or distance traveled by the robot.

```
(define
    (problem task)
    (:domain robot)
    (:objects
        wp0 wp1 wp2 - waypoint
        robot - robot
    )
    (:init
    (robot_at robot wp0)
    (= (distance wp0 wp1) 3)
    (= (distance wp0 wp2) 6)
    (= (distance wp1 wp0) 3.5)
    )
    (:goal (and
        (visited wp1)
        (visited wp2)
    ))
)
```

**Code 2.2:** PDDL problem for the robot

## 2.3 AI Planners

AI planners are the tools used for solving planning problems, generating a sequence of actions (a plan) to transition from an initial state to a goal state based on a defined model of the world. When fed with a domain description and problem instance, the planner will try to solve the problem and, if possible, output a list of actions that can be executed by an agent. How the planner solves the problem varies from planner to planner, and different planners will be efficient for different problems, depending on the method that the planner is based on.

As the field of planning and the Planning Domain Definition Language (PDDL) has evolved, so have AI planners. This means that different planners support different versions of PDDL, and not all features of a given PDDL version will necessarily be supported. This detail is important when choosing a planner, and using a planner that is able to handle all the needed PDDL features is therefore key when designing an AI planning system.

As described in section 1.3, planning algorithms and the inner workings of AI planners is not a part of the scope of this thesis. However, in order to provide some background for decisions discussed later in this thesis, some details of two different AI planners will be provided in this

section. These planners are the Partial Order Planning Forwards (POPF) [13] and Local search for Planning Graphs (LPG) [14] planners, and will be used to solve planning problems in this project.

### 2.3.1 Partial Order Planning Forwards (POPF)

The Partial Order Planning Forwards (POPF) planner is, as the name suggests, a planner based on a partial-order planning approach. With this approach, the planner uses the principle of least commitment and will therefor only commit to an ordering of actions when strictly necessary [15]. This approach is in contrast to total-order planning, where the order of all actions are determined and fixed. One of the advantages of partial-order planners is its ability to produce plans that are easily understandable by humans. This is an important feature for use cases where an operator is validating the plan before dispatch, such as in space exploration [6]. Achieving an optimal partial-order is a NP-hard problem [16] and the POPF planner does therefore not guarantee optimality, but is rather a planner that is usually able to find valid plans efficiently [13].

POPF planner was introduced by Coles, Coles, Fox, *et al.* [13] for the 2010 International Conference on Automated Planning and Scheduling (ICAPS 2010). Partial-order planning was a popular planning approach in the 1980s and 1990s and implemented in many popular planners of the time, such as UCPOP [17]. However, the development of other planning approaches in the early 2000s, like forward-search planners, outperformed partial-order planners significantly on fully automated classical planning problems [6]. Exploring the use of partial-order planning in a forward search framework, POPF showed significant improvements compared to normal partial-order planners [13]. POPF was created to support PDDL 2.1 and therefor handles the two new key features in PDDL 2.1: *durative actions* and *numeric fluents*. It does however not support features such as *negative* and *disjunctive preconditions*, and *conditional effects*.

### 2.3.2 Local search for Planning Graphs (LPG)

Local search for Planning Graphs (LPG) [18] in a PDDL2.1 planner that was introduced in 2002 and took part in the third International Planning Competition (IPC-3). LPG uses a local search, graph-based approach to solve both temporal and numeric planning problems, and is able to handle both plan generation and plan adaptation problems. The planner uses an *anytime algorithm* that iteratively generates progressively higher quality plans as it runs, making the planner able to return a valid plan at any given time during execution. LPG uses a compact representation of the planning graph and an evaluation function that estimates the search cost and execution cost of actions in the plan [18]. As the plan increases in quality, the complexity increases and the planner will spend more time on each iteration to find a higher quality plan. When a certain time limit is reach, the planner will terminate and the last created plan is output. The anytime planning approach provides a trade-off between plan quality and use of computational resources. LPG, like POPF, is a suboptimal planner that, when it exists, will provide a plan that is sound but not guaranteed to be optimal.

In 2004, *LPG-td* [14] was introduced as an extension and improvement to the original LPG planner, adding PDDL2.2 capabilities. The planner participated in the fourth International Planning

Competition (IPC-4).

The LPG-td planner can be run with different arguments, giving different behavior:

- *-speed*: Outputs the first plan that the planner finds.
- *-quality*: Finds a solution and iterates to find better solutions. CPU-time spent on improving the solution is automatically chosen.
- *-n <max number of desired solutions>*: Finds a solution and iterates to find better solutions. The number of desired solutions is an upper bound and might not be reached if maximum CPU-time is exceeded. (LPG-td only)

Both LPG and LPG-td has been influential in the planning community, and are well-known and widely used temporal planners. During IPC-3, LPG proved to be a highly diverse and powerful planner, solving 372 out of 428 (87%) attempted problems [19]. Later unofficial tests gave even better results, with 468 out of 508 (92%) problems solved [20]. The influence of LPG has also been shown by the *ICAPS 2019 Influential Paper Award* being awarded to the paper that introduced LPG in 2002[21]. In IPC-4, the LPG-td planner also performed well, being awarded second place in the suboptimal planner category [22].

## 2.4 Robot Operating System (ROS)

Contrary to its name, Robot Operating System (ROS) is not an actual operating system, but rather a free, open-source robotics Software Development Kit (SDK), used to build and develop robotics platforms [23]. ROS has become the de facto standard for robotics software in academia and many industries, resulting in a large community with free-to-use resources and libraries.

The fundamental concept of ROS is to be modular, and its architecture revolves around the use of nodes, topics, messages, and services. A language-neutral Interface Definition Language (IDL) describe the messages sent between modules and this allows ROS to be a multilingual platform with several supported programming languages, Python and C++ being the most commonly used [24]. This multilingual property gives the developer flexibility when writing code, in that they can choose the language that is best suited for the specific node in regard to efficiency, programming time and ease of debugging [23].

With a set of standard message classes, it is easy to implement new modules to the system, with standard message classes describing everything from single integers to the entire pose and attitude of an Inertial Measurement Unit (IMU). It is also possible to create custom messages when the standard messages do not provide the needed functionality.

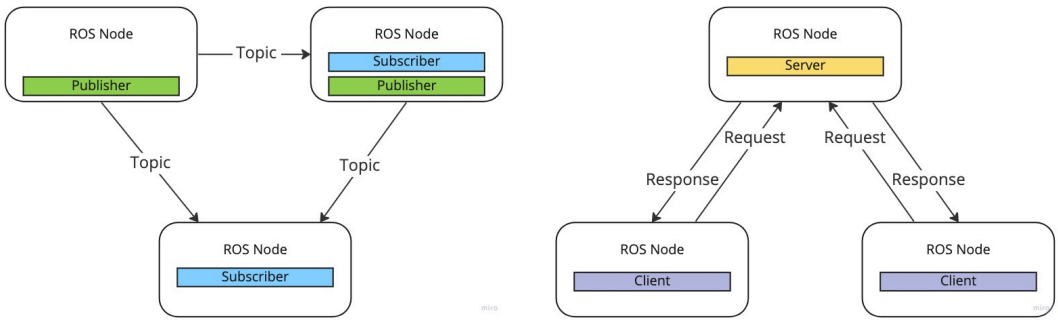
There are two major versions of the Robot Operating System, ROS 1 and ROS 2. ROS 1 [23] was released in 2007 and is the original version of ROS. ROS 2 [25] was released in 2017 and is the newest version. Overall, ROS 2 is a more modern and advanced version of the Robot Operating System, and is designed with the focus on making ROS a viable platform for commercial applications. Security, real-time systems and multi-platform support are key aspects where ROS 2 outperforms ROS 1 [25]. However, ROS 1 is still widely used and supported, and many ex-

isting ROS 1 applications can also be migrated to ROS 2 if needed. As ROSPlan and the other systems used in this project only supports ROS 1, the focus of in this thesis will only be on ROS 1 functionality.

ROS handles complex systems by decentralizing processes through the use of *nodes*. Nodes are coded programs that have specific responsibilities and perform its tasks without being aware of other nodes in the system. As mentioned earlier, these nodes can be written in different programming languages. For the different nodes to be able to communicate with each other, ROS uses a publish-subscribe model, in which nodes publish *messages* to *topics* and other nodes can subscribe to those topics to receive the messages. The messages can consist of various data types, such as integers, boolean values, and arrays. Nodes can both publish and subscribe to multiple topics.

In some cases, it may not be necessary to broadcast all values at all times. In these situations, ROS also offers a request-reply interaction called a *service*, in which a client node sends a request to a service node and waits for a reply. This can be useful for processes that only need to be performed occasionally, such as fetching parameters from a server.

Visualization of a ROS system is often done with a block diagram, where nodes are represented as boxes and topics and services are visualized by arrows between the nodes. It can also be represented as a graph, where nodes are represented as circles and topics are represented as boxes. Figure 2.2 shows a visualization of the publisher-subscriber and service-client concepts.




**(a)** Publisher-Subscriber **(b)** Service-Client

**Figure 2.2:** Visualization of ROS nodes, topics, and services.

### 2.4.1 Gazebo Simulator

Gazebo [26] is a simulation environment that is often used in conjunction with ROS. It is a well established tool and provides large amounts of features and libraries that allow users to simulate a virtual environment. For robotics, it allows the user to test a robotic system's behavior before deploying it in the real world. Gazebo provides a range of features that make it well-suited for robotics simulations. It has support for a wide variety of robots, sensors, and objects, and allows users to create and customize their own robots and environments. The gazebo_ros_pkgs package provides support for ROS, allowing users to easily integrate simulations with other ROS tools and libraries. [27]

Gazebo can be used in a variety of applications, including the development and testing of robotic

algorithms, the design and evaluation of robot behaviors, and the validation of robot designs. It provides a powerful tool for building and evaluating robot systems in a safe and controlled environment [27].

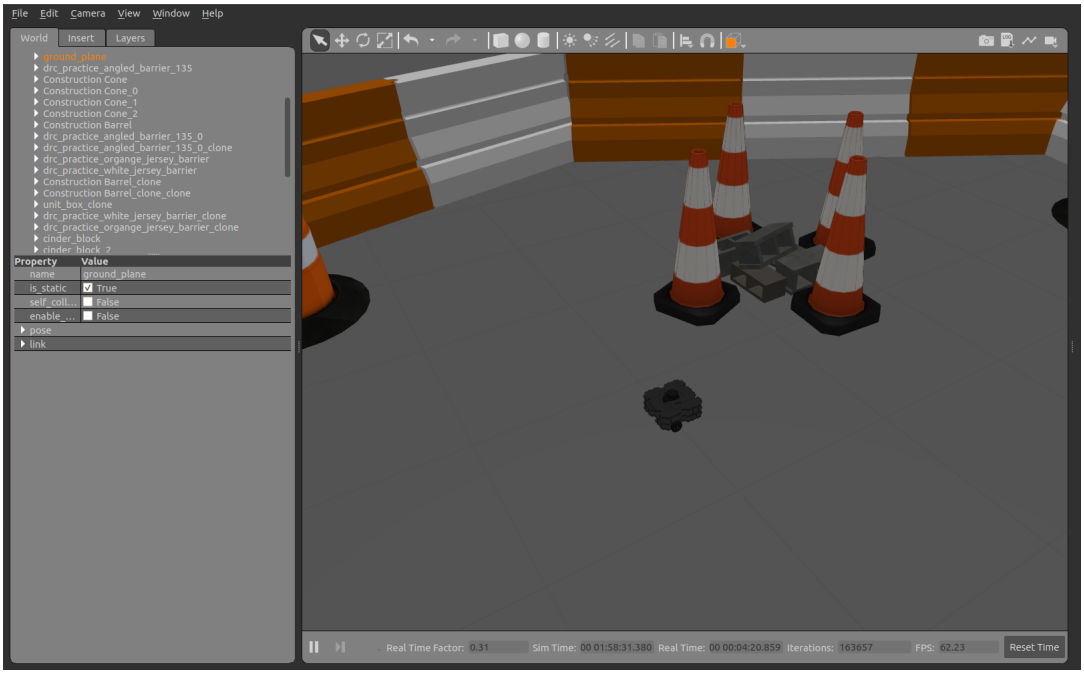



**Figure 2.3:** Screenshot of the Gazebo simulator, with a robot in a 3D modeled environment

## 2.5 AI planning with ROSPlan

ROSPlan [28] is a framework for building and deploying decision-making systems for robots. It provides a framework that integrates a ROS system with a task planner interface to create automatically generated plans of action for a robotic system. One of the key features of ROSPlan is its ability to dynamically store and update the state of the robot and environment as new information becomes available. This makes it particularly useful in uncertain or dynamic environments, where the robot needs to adapt to changing conditions [29]. ROSPlan uses an offline planning approach, meaning that the system computes an entire plan before executing it step by step. This is in contrast to an online planning approach, where the planning system is creating and executing plans concurrently, making decisions based on the current situation during dispatch.

The ROSPlan framework provides several tools and interfaces that when put together are able to generate and execute plans. This, as well as the nature of ROS, makes the framework highly modular and makes the user able to choose what parts of the system to include, depending on their needs. The architecture of ROSPlan can roughly be divided into two main pieces: the Knowledge Base and the planning system. The Knowledge Base is a key component of the ROSPlan framework and is responsible for storing information about the robot's states and environment, as well as the PDDL domain provided by the user. The planning system consists of several interfaces that process the data stored in the Knowledge Base [29]. Figure 2.4 illustrates the structure of the Knowledge Base and planning system of ROSPlan

### 2.5.1 ROSPlan planning system

The *problem interface* is the first interface in the ROSPlan planning system. When this interface is called through a ROS service, the interface queries the current state and PDDL domain in the Knowledge Base to generate a new PDDL problem instance. This problem instance is needed to let the planner know the current state of the system and to format the information into a PDDL format that the planner can understand. [28]

The *planner interface* is the ROSPlan interface that allows the system to use different external AI planners to solve the problem. The input of the planner interface is the problem instance created by the problem interface and the domain that is stored in the Knowledge Base. The interface is called by a service that returns true if planner was successful in generating a plan. The output of the interface is a sequence of actions that make up the plan, published as a string topic and saved to a file [28].

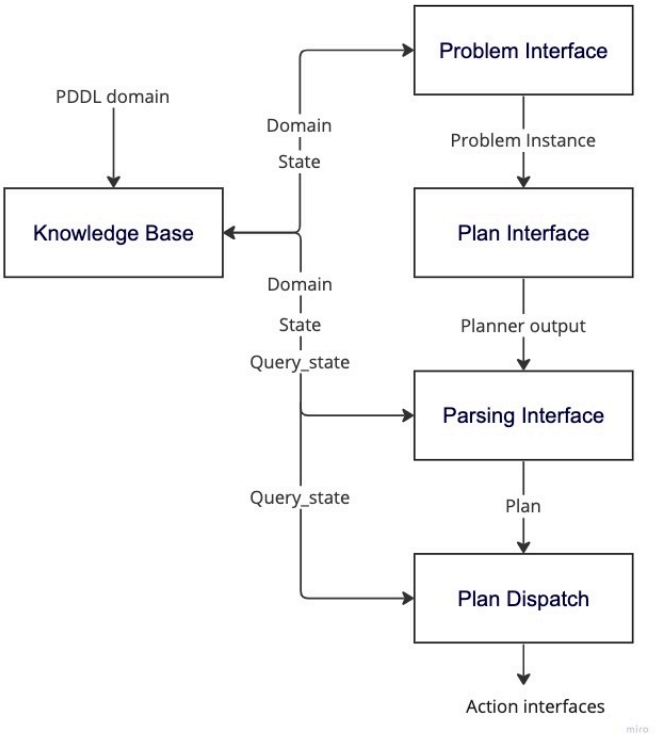



**Figure 2.4:** Chart illustrating the ROSPlan framework structure. Boxes representing ROS nodes and lines representing topics. Illustration based on figures from the ROSPlan documentation [28].

The ROSPlan framework does not provide its own AI planner. Instead, the planner interface feeds the needed information to external planners that is developed separately from the ROSPlan project. In theory, any PDDL planner should work with ROSPlan, as long as it is able to handle the requirements of the domain. According to the ROSPlan website [28] the planner interface has been implemented with several planners, providing different capabilities and limitations in regard to supported PDDL versions and computation capabilities.

To make the generated PDDL plans executable by a robotic system, the *parsing interface* is provided as part of the planning system. This interface is able to represent the planner output with different structures, depending on what type of execution that is wanted. The *simple plan* structure is sufficient for sequential dispatch or temporal plans, where the actions are executed when the previous action is completed, while the *esterel plan* is recommended for plans with concurrent execution [28].

The last part of the planning system is the *plan dispatch*. This interface is tasked with taking care of the execution of the plan by connecting the high-level control of ROSPlan to the low-level control of the robotic platform. The actions to be executed will be published to an action_dispatch topic.

The process of reacting to these dispatch messages is then taken care of by an action interface that has to be customized to the specific robotic system [29].

When an executed action fails, an update to the Knowledge Base invalidates the plan, or the currently executed action has passed a timeout threshold, the plan dispatch interface will stop the execution of the plan. ROSPlan will then require a re-planning by reformulating the problem based on the new information [29]. Re-planning functionality does not come pre-made in ROSPlan and will have to be implemented by the user.

## 2.6 Relevant literature overview

This section aims to highlight a selection of research papers covering similar topics and projects to what is covered in this thesis. This should provide a better understanding of the contribution of this thesis, as well as provide some insight into the scope and subjects touched upon in the project.

As mentioned in section 2.1, planning has been a field of research since the 1970s, starting with the STRIPS project, and research on classical planning problems continued through the 80s and 90s [7]. However, it is only more recently that AI planning has become a more relevant approach for use in real-life autonomous robotic mission. As the field of research has matured, many use cases for AI planning have been proposed, for example underwater robotics [30]–[33], space exploration and satellite applications [34]–[37], and robot-human interaction [38]–[40]. Although many use cases are being proposed in the literature, it is challenging to find concrete examples of implementations that are more than that of theoretical or experimental nature. One of the areas where the use cases of AI planning and scheduling have received most attention is the space industry, most notably for the use of AI planning in the Mars Exploration Rover project [3], [35]. In this project, AI planning was used to enable planning and scheduling of daily activity plans for the rover, and is one of the most exciting examples of uses of planning in robotics.

The focus of this thesis is on AI planning using the Robot Operating System (ROS) and the ROSPlan framework, and several examples can be found of ROSPlan being used for autonomous robotics with AI planning. Xue and Lekkas [32] presents a comparison of two different AI planning frameworks for underwater intervention drones, ROSPlan [29] and T-REX [41]. They found that the non-ROS-based T-REX framework generally worked better than ROSPlan in dynamic environments, mainly due to ROSPlan doing offline planning and T-REX doing online planning.

The work of Hoteit, Abdallah, Faour, *et al.* [39] demonstrates the application of ROSPlan for autonomous operations of a social assistive robot. It details the use of ROSPlan in the creation and execution of plans and provides an overview of how these plans are emulated through a TurtleBot3 robot simulation in Gazebo. The Partial Order Planning Forwards (POPF) planner [13] is used to solve the planning problem and two algorithms are proposed to enable re-planning in the event of a plan failure, one of the algorithms being incorporated into the simulation. Additionally, the paper touches on some of ROSPlan's limitations, specifically for only supporting temporal planning, while probabilistic or conditional planning remains unsupported. The paper describes a system that is in many ways similar to the system used in this thesis. Both the described use of ROSPlan and robot simulator in Gazebo is similar to many of the methods used in this project and highlights

how the modular design of ROS and the ROSPlan framework allows a general solution to be used for a diverse set of use cases.

Sanelli, Cashmore, Magazzeni, *et al.* [38] is another example of human-robot interaction robots that utilize ROSPlan. The authors propose a method and present a fully implemented robotic system that utilizes conditional planning, using ROSPlan and the Petri Net Plans execution framework, to generate and execute short-term human-robot interactions. The implementation was successfully tested in different scenarios where the robot interacted with untrained users.

Overall, the literature on AI planning is extensive, and many use cases has been proposed. However, finding examples of implementation of AI planning in real-world applications has proved difficult, and from the few cases found in this literature search, most were used for very special cases, like space related use cases, where constraints and limitations are very different from the once present in this project. It seems from this apparent that the field of AI planning has not reached its potential for autonomous robotics, and that work still remains in order for autonomous AI planning systems to be deployed in real-world applications.

# 3

# PDDL domain description and problem instance design

This chapter describes how the domain description and problem instance was designed in an effort to answer the research questions in section 1.2. The domain description should be a general representation of the environment and actions available, enabling it to be scalable and used for different missions and robots. The problem instance is the representation of a specific mission, and the problem presented in this chapter is therefore an example of how the problem can be defined to perform an mission like described in section 1.1, with objects of interest that are to be inspected and a robot with limited battery capacity.

The ROSPlan framework supports PDDL 2.1 [12] and this, with its features and constraints, forms the basis for the PDDL design for this project. In the following sections, each part of the PDDL domain and problem design is explained. Together with the explanation, code snippets are provided to show how the code is formatted. More details about the Planning Domain Definition Language (PDDL) has been described in section 2.2. The full PDDL code described in this chapter is found in appendix A and B.

## 3.1 Domain description

The PDDL domain describes in this case a robotic system, designed for Inspection and Maintenance (I&M) missions. The capabilities needed from the domain in order to plan the mission described in section 1.1 are:

- A robot moving to specified waypoints.
- Robot taking a photo at waypoints specified as objects of interest.
- Include battery into the planning problem, making it able to account for battery usage.
- Directing the robot to a charger when battery is low.
- The robot docking to the charging station.

As AI planning only handles the high level control of the robotic system, it is important to emphasize that there is a difference between the actions in the domain and the capabilities needed by the robot to perform this task. For the list above, a robot would typically need:

- Hardware and software to move around the environment autonomously.
- A camera capable to take inspection photos.
- Charging station and capability to dock to it.

### 3.1.1 Types, predicates, and functions

The domain for this project defines two objects using the *typing* requirement. These objects are

- ***waypoint***, locations on the map that are used when navigating
- ***robot***, making it possible to plan with different, or several, robots in the same domain.

```
(:types
  waypoint
  robot
)
```

Five *predicates* are used to describe the state of the system:

- ***robot_at*** describes where the robot is in the environment at any given time
- ***docked*** and ***undocked***, used in connection with the robot charging, describing whether the robot is docked or undocked to a charger
- ***charge_at***, describing the location of charging stations
- ***photographed***, describe what objects of interest have been photographed

```
(:predicates
  (robot_at ?v - robot ?wp - waypoint)
  (undocked ?v - robot)
  (docked ?v - robot)
  (charge_at ?wp - waypoint)
  (photographed ?wp - waypoint)
)
```

One of the advantages with the way a PDDL planning problem is designed is that the domain description can be designed very generalized, making it possible to apply it on various problems of different scales and complexity. In the domain design described here, an effort to generalize as much as possible was made. This was done because a specific robot or operating environment was not defined in the mission. To achieve this generalization, extensive use of the numeric *functions* feature in PDDL2.1 was used. This made it possible to change parameters of the robots in the problem instance used for the specific mission. Eight functions are used to describe the system:

- ***distance***, describes the distance between two waypoints
- ***speed***, describes the movement speed of the robot, allowing different robots with different speeds to be defined in the same domain
- ***min_charge***, describe the minimum battery charge allowed at any point

- ***state_of_charge***, current battery charge of the robot
- ***charging_rate*** and ***discharge_rate***, makes it possible to define different battery charge and discharge rates for different robots
- ***docking_duration***, defines the duration a robot uses to dock to a charger
- ***traveled***, describes the distance a specific robot has moved. This function can be used as a metric in the problem instance to allow the planner to minimize distance in its plan.

```
(:functions
  (distance ?wp1 ?wp2 - waypoint)
  (speed ?v - robot)
  (min_charge ?v - robot)
  (state_of_charge ?v - robot)
  (charging_rate ?v - robot)
  (discharge_rate ?v - robot)
  (docking_duration ?v - robot)
  (traveled ?v -robot)
)
```

### 3.1.2 Actions

For the inspection robot to be able to perform its missions, the most important *action* needed is to move around in its environment. In this domain, the *goto_waypoint* action is designed as a temporal action with a start and end waypoint. The duration of the action is defined as `:duration (= ?duration (/ (distance ?from ?to)(speed ?v)))`, translating to $duration = \frac{distance}{speed}$. This ensures that the duration of the action can be adapted to the specific robot used for the mission by defining different speeds for different robots in the problem instance.

For the action to be applicable, three conditions are defined. As it is a durative action, the prefix of the condition determines at what point in the action they should be fulfilled. The conditions for the *goto_waypoint* action are: (1) The robot must be at the starting waypoint, (2) the robot must be undocked for the whole duration, and (3) *state_of_charge* must never go below *min_charge*. The function in condition (3) can be translated to

$$state\_of\_charge - distance * discharge\_rate \geq min\_charge$$

at the start of the action, ensuring that the battery charge will be above min_charge when reaching the end waypoint. These three conditions ensures that the robot is in a state where it is able to move as desired, and that the battery is never discharged below the minimum level specified in the problem instance.

Four effects are defined for this action. (1) At the start of the action, the robot is no longer at the starting waypoint, (2) at the end of the action, the robot is at the end waypoint, (3) *state_of_charge* is decreased according to the distance it has traveled, and (4) *traveled* function is increased by the distance the robot has traveled. From the perspective of the Knowledge Base, the robot now has been moved from starting waypoint to end waypoint, battery level has decreased, and the total distance traveled has increased.

```
; Move robot between waypoints
(:durative-action goto_waypoint
  :parameters (?v - robot ?from ?to - waypoint)
  :duration (= ?duration (/ (distance ?from ?to)
                (speed ?v)))
  :condition (and
    (at start (robot_at ?v ?from))
    (over all (undocked ?v))
    (at start (>= (- (state_of_charge ?v)(* (discharge_rate ?v) (distance ?from
     to)))(min_charge ?v)))
    )
  :effect (and
    (at start (not (robot_at ?v ?from)))
    (at end (decrease (state_of_charge ?v) (* (discharge_rate ?v) (distance ?
    from ?to))))
    (at end (robot_at ?v ?to))
    (at end (increase (traveled ?v) (distance ?from ?to)))
    )
)
```

The *dock* and *undock* actions are related to, as the name suggests, the robot docking to a charging station. These durative actions are oppositions of each other and therefore very similar in design. The duration of the actions is defined by the *docking_duration* function that allows for defining different docking durations for different robots. Three conditions are defined for the actions: (1) the robot should always be on the defined waypoint, (2) the robot should be on a charger, defined by the *charge_at* predicate, and (3) the robot should be undocked for the *dock* action or docked for the *undock* action. The effects of the actions adds or removes the *docked* and *undocked* predicate.

```
; Docking to charger
(:durative-action dock
  :parameters (?v - robot ?wp - waypoint)
  :duration ( = ?duration (docking_duration ?v))
  :condition (and
    (at start (charge_at ?wp))
    (over all (robot_at ?v ?wp))
    (at start (undocked ?v))
    )
  :effect (and
    (at end (docked ?v))
    (at start (not (undocked ?v))))
)
```

The charging action is defined to enable the battery simulation and is the only way *state_of_charge* can increase. As with the other actions, the charge action is defined as a durative action. The duration of the action is determined by the state of charge, simulating how the charging duration of a real battery varies depending on the initial battery level. Similar to the discharge rate in the goto_waypoint action, the duration of this action is defined with a *charging_rate* function that can be tuned to the specific robot. Identically to the *undock* action, the conditions of the charge action demand that the robot is (2) located and (3) docked at a waypoint that it is (1) defined as a charger. The effect of the charging action is that the state of charge is assigned a value of 100%.

```
; Charging battery
(:durative-action charge
  :parameters (?v - robot ?wp - waypoint)
  :duration ( = ?duration (* (charging_rate ?v - robot) (- 100 (state_of_charge
     ?v))))
  :condition (and
    (at start (charge_at ?wp))
    (at start (robot_at ?v ?wp))
    (over all (docked ?v)))
  :effect (and
    (at end (assign (state_of_charge ?v) 100))
    )
)
```

Lastly, the *inspect* action simulates the robot inspecting an object of interest. The duration of the action is arbitrarily set to 10 seconds, but this could be changed depending on what an inspection mission involves for the specific robot. This could also be defined using a numeric function, like has been done in the other actions, this is however not done here for simplicity. The conditions for the actions are: (1) the robot has to be at the defined waypoint and (2) the state of charge should be higher than *min_charge* when the action is performed. The effect of the action is that (1) the defined waypoint is inspected, in this case by the *photographed* predicate, and a decrease of the state of charge. The reason for this action decreasing the state of charge is that one could imagine that the act of inspecting an object would discharge the battery slightly. The magnitude of this discharge should be tuned depending on the inspection task, and could therefore also be defined as a numeric function.

```
; Photographing an object of interest
(:durative-action inspect
  :parameters (?v - robot ?wp - waypoint)
  :duration ( = ?duration 10)
  :condition (and
    (over all (robot_at ?v ?wp))
    (at start (>= (- (state_of_charge ?v) 3) (min_charge ?v))))
  :effect (and
    (at end (photographed ?wp))
    (at end (decrease (state_of_charge ?v) 3))
    )
)
```

## 3.2 Problem instance

The problem instance is a representation of a specific problem or mission. This means specifying objects, initial conditions and goals, as opposed to the domain description where the system is described with general variables. As the problem instance is defined for specific missions, the instance described here is an example of how a problem can be defined for the domain presented above, based on the mission described in section 1.1.

The mission is defined to contain eight waypoints and one robot, however as described earlier in this chapter the domain is designed to make it possible to define several robots if desired.

```
(:objects
    wp0 wp1 wp2 wp3 wp4 wp5 wp6 wp7 - waypoint
    turtlebot - robot
)
```

The *initial conditions* for the mission describe everything that is true at the start of the mission. As mentioned in chapter 2, PDDL is based on a *closed-world assumption* by default. This means that negative predicates do not have to be defined in the initial conditions, as undefined predicates are assumed to be false. In the example below, the robot is initialized at waypoint 0. It is also undocked. The *charge_at* predicate that describes what waypoints the chargers are located at are set to waypoints 0 and 1.

As the domain uses distance between waypoints in several actions, all distances have to be defined in the problem description. In the example below, only a few of the distances are included in order to save space in the report. Note that the distances has to be defined twice (e.g., wp1 - wp0 and wp0 - wp1) to make it possible to go both ways between the waypoints. Due to the closed-world assumption, distances that are not declared should ideally not be considered by the planner. However, different planners handles numeric functions differently, and some planners will therefore handle undefined function values as equal to zero, necessitating all distances to be declared.

```
(:init
    (robot_at turtlebot wp0)
    (undocked turtlebot)
    (charge_at wp0)
    (charge_at wp1)

    (= (distance wp1 wp0) 5.59464)
    (= (distance wp0 wp1) 5.59464)
    (= (distance wp2 wp0) 2.94109)
    (= (distance wp0 wp2) 2.94109)
    (= (distance wp2 wp1) 5.80086)

    ; The rest of the distances omitted to save space

    (= (speed turtlebot) 0.1)
    (= (min_charge turtlebot) 15)
    (= (charging_rate turtlebot) 0.5)
    (= (discharge_rate turtlebot) 3)
    (= (docking_duration turtlebot) 1)
    (= (traveled turtlebot) 0)
    (= (state_of_charge turtlebot) 100)
    )
```

The last part of the initial conditions define the different numeric functions used in the domain, all parameters related to the robot used in the mission. In this example, values have been chosen to give the desired behavior for simulated missions described later in the thesis. These would need to be tuned when implemented on a specific robot.

The *goals* in this example are chosen to represent a mission where all six objects of interest (all waypoints that are not defined as chargers) should be photographed. Lastly, the *traveled* function is defined as a minimization metric. This ensures that the planner tries to find the shortest path to

the goal. Other metrics could also be chosen, for example, total mission time.

```
(:goal (and
    (photographed wp2)
    (photographed wp3)
    (photographed wp4)
    (photographed wp5)
    (photographed wp6)
    (photographed wp7)

))
(:metric minimize (traveled))
)
```

Summarizing this example problem instance, we have an environment with eight waypoints and one robot. The robot is to photograph six objects, traveling the shortest distance possible. Due to the *min_charge* predicate, the battery level of the robot should never go below 15%. If needed, the robot can charge its batteries at either waypoint 0 or 1. As stated earlier, the problem description presented above is only an example of how it could be designed. In practice, different problems could be defined in many different ways with the same domain description, showing one of the strengths of the Planning Domain Definition Language (PDDL).

# 4

# ROSPlan feature extension and simulator setup

This chapter describes how the ROSPlan framework was set up to fit the purpose of this project. The chapter describes both the setup process and design decisions made, as well as changes and extension made to the stock configuration of the framework. The chapter also describes the simulator setup for the project and is, in large part, taken from the report of the specialization project [4] of fall 2022. Lastly, the setup of the lab experimental tests are described.

The ROSPlan framework is provided through a repository on the official King's College London (KCL) Planning GitHub [42]. The repository contains code for the ROSPlan Knowledge Base and all the interfaces needed for planning and execution of plans. These interfaces are described in more detail i Section 2.5. The GitHub repository, as well as the ROSPlan website [28], provides in-depth documentation and tutorials that are useful when setting up and running ROSPlan for the first time.

In addition to the ROSPlan package, KCL-Planning also provides a demo repository [43] for ROSPlan. This repository contains several demos with example code for running ROSPlan. Notably, this repository includes a TurtleBot3 exploration mission demo, wherein the robot is tasked with traversing various waypoints within a simulated Gazebo environment. Despite its simplicity and limited functionality, this demo serves as an initial integration between ROSPlan and TurtleBot3, incorporating simulation in Gazebo.

## 4.1 ROSPlan Knowledge Base

The ROSPlan Knowledge Base is the node that stores all information about the system. This information can then be used to create a PDDL planning problem that is sent to the AI planner. When the system is launched, the Knowledge Base is fed PDDL domain and problem files. The domain describes the environment of the system, and the problem describes the specific mission. These files are provided to the Knowledge Base as .pddl files of the structure described in chapter

3. With this information, and other information that is fed to the Knowledge Base through ROS services, a new PDDL problem instance can be created by the Problem Interface. This problem instance can then be fed to the AI planner through the Planning Interface.

As the Knowledge Base can be updated through ROS services, it can always be kept up to date, and a new problem instance can be crated at any point during the mission. This makes it possible to re-plan a mission with updated information when needed.

## 4.2 Planner implementation

As part of the ROSPlan package, the Planner Interface is provided. ROSPlan does not provide AI planning capabilities itself, instead the Planner Interface is a wrapper that allows for the use of different AI Planners that are fit for the specific problem. As the input and output format of different AI Planners differ, each planner needs a custom interface to be able to handle the parsing of information. ROSPlan comes with several pre-installed planner interfaces for different AI planners. This forms the basis for the choice of planners for this project, as it reduces the work needed to implement the planners. The ROSPlan website [28] lists implemented planner interfaces for these planners:

- POPF
- OPTIC
- Fast Forward (FF), Metric FF, Contingent-FF
- LPG
- TFD
- SMTPlan

In addition to the planners listed on the website, more planner interfaces are provided in the ROSPlan source code. Examples of these are *Fast Downward (FD)*, *PANDA*, *CHIMP*, *RDDLSim*, and *UPMurphi*.

As the focus of this project is not specifically targeted towards planning and AI planners itself, but rather the tools and methods needed around the planner, the only requirements for the choice of AI planners are that they are easy to implement, able to handle all needed PDDL requirements, quick, and accurate.

The *POPF* planner executable comes pre-installed as the default planner in ROSPlan, and including it as one of the planners used in the project was therefore natural. The *Fast Forward (FF)* planner is a well-known family of planners, however the original FF planner does not support temporal or numeric problems. *Metric FF* extends FF to support numerics, but still does not support temporal planning that is needed for this project. Neither does Contingent-FF. In addition to POPF, the *LPG* planner was also chosen to be used for the project. As described in section 2.3, LPG is a popular and well-known temporal planner, able to handle all necessary PDDL2.1 requirements. For this project, the newest version of the LPG planner is used, LPG-td [14].

Finding, downloading, and compiling AI planners generally proved to be difficult. Many of the

planners listed above are old planners from the early 2000s, leading to many of the planners, or dependencies, being deprecated and causing issues when compiled. OPTIC, SMTPlan, and TFD are examples where this was a problem. The problem of obtaining and setting up AI planners is a known problem in the AI planning community. The Planutils project [44] from 2022 aims to resolve this problem by providing a tool for developing, running, and evaluating planners. The Planutils tool was not tested for this project, but implementing it into the ROSPlan framework could be a potential for further work, in order to enable easy access to more AI planners.

## 4.3 Actions Interface

As ROSPlan only handles high level control, the system needs a way to connect to the lower level control of the robot in order to perform the generated plan. ROSPlan handles this is with the help of the Action Interface. The action interface subscribes to the action dispatch topic, published by the Plan Dispatch node, and sends commands to the lower level control. As this involves communicating with systems outside the ROSPlan framework, the action interface will need to be adapted to the specific robot in question. The ROSPlan tutorial [28] suggests two different possible approaches for solving this: Either extend and adapt the action interface that comes pre-made with ROSPlan, or implement an interface from scratch. A custom-made action interface is suited for systems where there is needed more control over when and how the Knowledge Base is updated, for example if the position of the robot is updated by sensors instead of add and del effects in the domain. For the purpose of this project, the pre-made action interface satisfied the needs with minimal changes. The only change made to the interface was adding PDDL functions *increase* and *decrease*, which was not originally implemented.

In addition to subscribing to dispatch messages and sending commands to the robot, the action interface also receives feedback from the robot in order to report action status back to the system. This information should be provided from the robot in the form of a success/failed action feedback message. For the system presented in this project, this is done by the robot for the *goto_waypoint* action. However, as the rest of the actions are not actually implemented on the robot, these actions have to be simulated. This can be done using the Simulated Action Interface that comes with the ROSPlan framework. This interface facilitates the simulation of action success or failure by communicating with the Action Interface, which in turn updates states in the Knowledge Base.

Upon launching a Simulated Action node, the node is configured with a corresponding action from the PDDL domain. The node then listens for an action dispatch call for this specific action, published by the Plan Dispatch interface. When the dispatch call is published, the Simulated Action node communicates the action's success or failure, based on its predefined settings. A key advantage of the Simulated Actions node is the ability to customize action duration and success probability, enabling the simulation of temporal actions and actions with uncertain outcomes.

It is crucial to emphasize that the Simulated Action node does not execute any actions, it merely simulates the success or failure of an action and reports this information to the Knowledge Base. Consequently, an alternative solution must be devised when transitioning the system to a real-world context.

## 4.4 Planning node

The next two sections describe features developed for this project in order to extend the functionality of the ROSPlan framework. The features were developed and implemented into the ROSPlan code, forked to the author's GitHub repository [45]

Originally, the ROSPlan framework does not come with functionality that allows the user to run the different interfaces in the planning sequence automatically. This functionality is not only needed when initiating a mission, but also when re-planning a mission is required. Because of this, a planning node was design to take care of all functionality related to planning and re-planning. In the ROSPlan TurtleBot3 demo, the planning interfaces are run in succession by running an included bash script. This bash script formed the basis for the planning node Python script. A flow chart of the planning node can be seen in Figure 4.1.

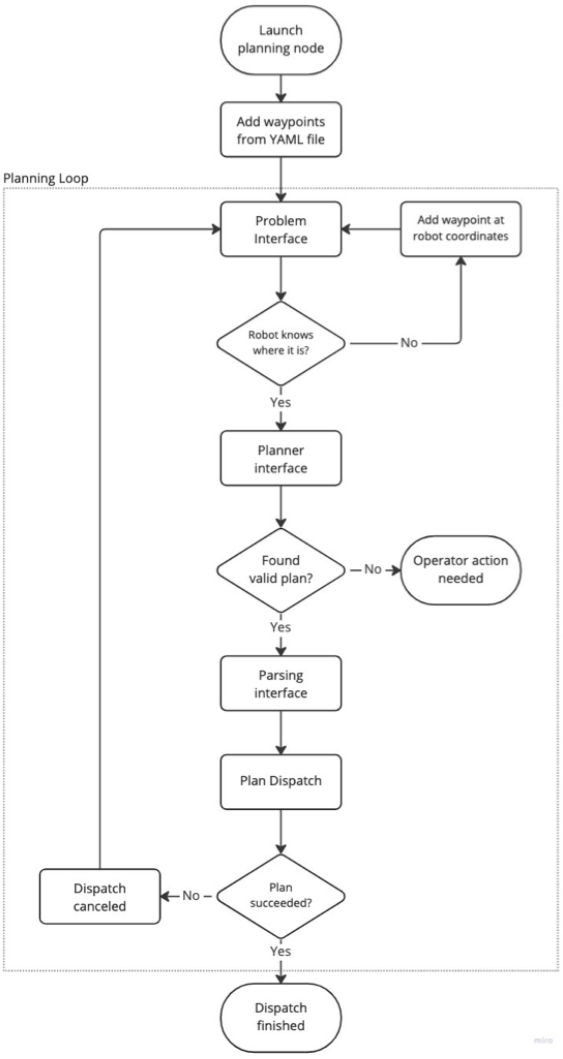



**Figure 4.1:** Flow chart illustrating the planning node with the planning loop created to allow for re-planning missions.

On initialization, the node will first add all waypoints, as well as initial states related to the waypoints, to the ROS parameter server and ROSPlan Knowledge Base. The Roadmap Server, included with the ROSPlan demos package, takes care of both adding the waypoints to the parameter server and Knowledge Base, as well as connectivity between the waypoints. The result is that the waypoints and distances between them is automatically added to the problem description when the Problem Interface is called. When the waypoints have been added, the node enters the planning loop. This loop runs all the ROSPlan interfaces mentioned in Section 2.5.

### 4.4.1 Unknown robot position

The first step in the loop is the Problem Interface that automatically generates a PDDL problem description from the information that is stored in the Knowledge Base. The next step in the planning sequence is now for an AI planner to generate a plan with the Planner Interface. However, as the planner needs to know the initial position of the robot in order to generate a valid plan, a

problem will occur if the robot has an unknown position. This issue is an inherent result of how a *goto_waypoint* action is designed, relying on both start and end position of the robot. The issue of the robot not knowing where it is usually occurs when a plan dispatch is canceled while the robot is between two waypoints, as the position of the robot is only updated in the Knowledge Base at the start and end of the action

To solve this problem, the node checks the generated problem description file for a *robot_at* predicate in the initial state. If no such predicate is found, some action is needed before running the Planner Interface. Several solutions for this problem were considered when designing the node. One solution was to redirect the robot to a predefined waypoint before running the planner. This waypoint could for example be a charger or docking station. It was a simple but less than ideal solution to the problem, as this would add significant time and distance to the mission, especially in a large environment. Several predefined waypoints could be added to the domain to reduce the distances between them, but this solution would require additional logic to decide what waypoint to choose, as well as adding more computational complexity by increasing the planning problem size. A similar solution considered was to redirect the robot to its last waypoint, with some of the same drawbacks of added distance and time to the mission.

The solution implemented in the planning node was instead to create a new waypoint at the coordinates of the robot. This solution was implemented through the ROSPlan roadmap server, which provides a ROS service that can add new waypoints at any time. This solution eliminates the need for the robot to spend extra time and energy to go to a known waypoint. One drawback of the solution is the added complexity of the planning problem when adding new waypoints, especially in the case of several instances of new waypoints being added. This problem could potentially be solved by removing the waypoint from the Knowledge Base after the re-planning, making sure that this waypoint is not included in the next re-planning.

### 4.4.2 Planning and re-planning

When the position predicate in the problem description is known, the planning loop can continue to the Planning Interface. This interface passes the PDDL domain and problem description to the chosen AI planner and outputs a plan. If the planner is not able to create a valid plan, the loop is stopped for the operator to solve the issue. Some examples of reasons a planner may fail are syntax errors in the domain or problem, a valid plan does not exist for the specific problem, or the planner not being designated enough CPU-time. These issues are difficult to resolve automatically, therefore needing an operator to resolve the issue. If the plan is valid, the sequence continues on to the Parsing Interface and Plan Dispatch.

During plan dispatch, the mission might fail for different reasons, like failure to reach a target due to a blocked path, or simulated failure of inspection. When an action fails, re-planning will be necessary. In this case, the planning node will automatically detect the failed action and cancel the dispatch. When the dispatch is canceled, the end of the planning loop is reached and as the plan did not succeed, the loop will start from the top as illustrated in Figure 4.1. This solution is simple, yet elegant, as it also handles manual dispatch cancellations through the *cancel_dispatch* service.

### 4.4.3 Goal removal

With the system described above, the re-planning sequence will always include the goal that failed. This might be desirable as the reason for the failed action could be due to factors that could be resolved with a new try, however if the problem persists, the system would get stuck in a loop of unsuccessfully trying to achieve the goal. The plan would always try to reach the goal, even though it is impossible to reach, causing a new re-planning every time. To prevent this, a system was developed to allow the planning node to remove goals that have failed several times. This feature works by registering and storing the number of times an action has failed. If the action failure count reaches a threshold set by the operator, the goal corresponding with this action is removed from the Knowledge Base before the re-planning sequence is run.

## 4.5 Operator interaction

One of the goals of the system described is to reduce the need for human interaction with the robot. However, some form of human supervision and control is usually always needed to ensure efficient and safe operation of a robotic mission. In this project, a simple system for operator interaction is proposed to enable validation of the inspection data. This feature is not meant to be a complete operator control system, but rather an example of a part of an operator-in-the-loop system, integrated into the autonomous mission.

The main tasks of the Inspection and Maintenance (I&M) missions in this project is to inspect objects of interest by photography. The resulting photos need to be evaluated in order to decide if further actions are needed. The operator node created for this project enables the operator to do this without directly interrupting the mission dispatch. Every time the robot inspects an object, the operator is prompted in the terminal window. This prompt allows the operator to either reject the photograph by pressing ENTER, or approve the photo by not taking any action. While the operator is prompted, the robot continues its mission, keeping the flow by preventing long pauses for the operator to inspect the photograph.

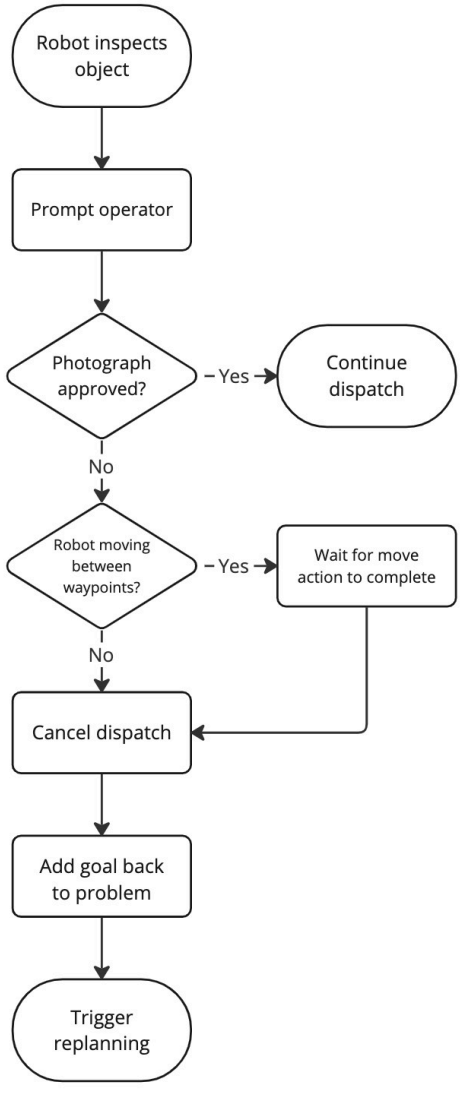



**Figure 4.2:** Flow chart illustrating the operator interaction node.

If the operator decides that the photograph is approved, the mission continues without interruption. However, if the operator decides that the photograph is not approved, for example because the photo is of too low quality, a dispatch cancellation is issued. If this cancellation is issued while the robot is moving between waypoints, the node waits until the action is completed and the robot is at the new waypoint, the dispatch is then canceled and the inspection goal is added back into the problem instance. A new iteration of the planning node can now be triggered to perform a re-planning. The reason for avoiding dispatch cancellation during a move action is to keep the flow of the mission and giving a more predictable behavior for people interacting with the robot.

It is important to note that in this project, the focus has been on human operator interaction. However, in an effort of increasing the autonomy of the robot further, the task of validating photographs could be performed by image processing and artificial intelligence. The operator interaction feature could also be increased with functionality like adding and removing goals, as well as handling dispatch errors like blocked paths.

## 4.6 Simulator environment

The simulator setup in this project is based on the work done in the preparatory specialization project [4] in fall 2022. The contents of this section are therefore reused from the project thesis, with some small modifications to section 4.6.3 due to changes done to the world model to fit the purpose of this project.

### 4.6.1 TurtleBot3

For teaching, demonstration and development purposes, the TurtleBot3 is developed and sold in a partnership between Open Robotics and ROBOTIS INC. [46]. The TurtleBot3 is the third generation of the small, low-cost ROS-based robotics platform. It is customizable to accommodate the needs of the developer, and a Gazebo simulator environment makes it possible to simulate the TurtleBot3 without the hardware. For this project, a TurtleBot3 simulator was set up to facilitate testing and validation of mission plans and systems.

The TurtleBot3 series consists of three different robots, the Turtlebot3 Burger, TurtleBot3 Waffle, and TurtleBot3 Waffle Pi. This project will focus on the TurtleBot3 Waffle Pi, the most advanced of the three. The Waffle Pi is powered by a Raspberry Pi as its Single Board Computer (SBC) and a 32-bit ARM Cortex®-M7 Microcontroller Unit (MCU). The TurtleBot3's main capabilities are within Simultaneous Localization And Mapping (SLAM), navigation and manipulation. For sensing and situational awareness, the Waffle Pi is equipped with a 360 LIDAR, a forward facing camera, and an Inertial Measurement Unit (IMU).

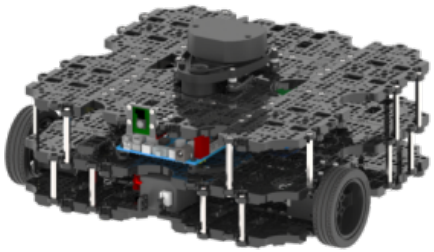


**Figure 4.3:** The TurtleBot3 Waffle Pi robotic platform [46].

Together with the robotics platform, an expansive open-source ROS package is provided for the TurtleBot3. This package comes with ready-made SLAM and navigation functionalities. A Gazebo simulation package is also provided for the TurtleBot3, making it easy to set up and run simulations of the robot. These functionalities will be used throughout this project when testing PDDL plans and system functionality.

### 4.6.2 Gazebo simulator

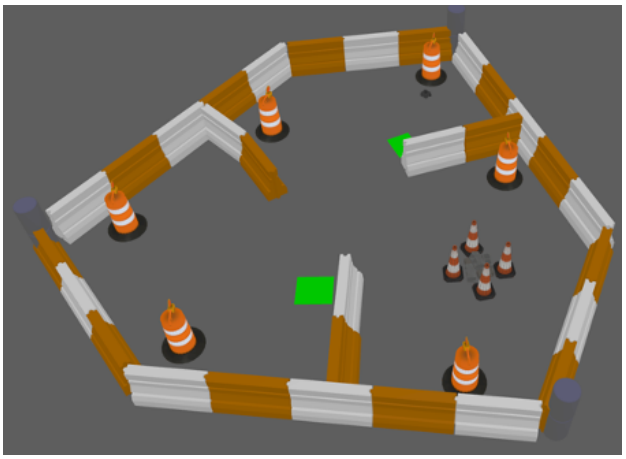
Setting up a TurtleBot3 simulation in Gazebo is fairly straight forward. Using the TurtleBot3 Simulations package [47], a TurtleBot3 3d model and the needed code is provided to facilitate simulation of a TurtleBot3 in Gazebo. As Gazebo is able to simulate sensor data like LIDAR, camera, and IMU, it is able to provide highly realistic simulated behavior with much of the same code that is run on a physical TurtleBot3. This means that there in theory should not be necessary to do large changes to the code when moving from the simulated environment to a physical system.

### 4.6.3 World model

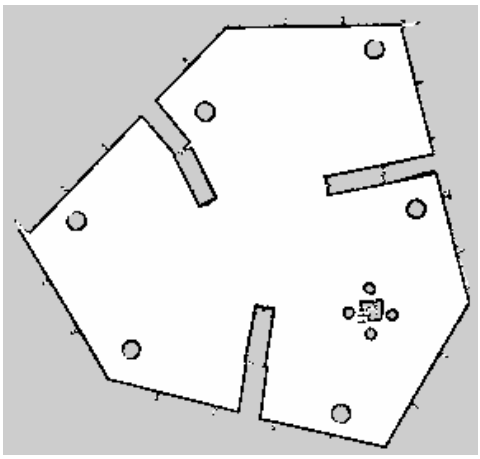
When running the TurtleBot3 simulator, the 3d model of the robot is spawned into a simulated 3d world. The TurtleBot3 Gazebo package comes with several pre-made 3d world models. These are worlds with different objects that the robot can interact with and showcases the abilities of the TurtleBot3 robotic platform. However, none of these pre-installed worlds fitted the purpose of this project, and a new 3d world model was therefore created. The world model was created using the Gazebo Graphical User Interface (GUI) and pre-made 3d object models. The design of the world was chosen to be easily changed, to add functionality, and to increase or decrease difficulty for the robot simulation when needed.

As seen from Figure 4.4a, the custom designed world consists of several obstacles and features that the robot can interact with. In this version of the model, there are six large orange cones that simulate objects of interest. These cones are meant to be investigated/photographed by the robot and could in a real world scenario represent pipes, gauges, or valves that operators need inspected. In addition to the objects of interest, two green zones are placed in the world. These zones represent charging stations for the robot. This is a simplified representation of a charger, and the robot simply has to be inside the zone for the battery to charge. For a real world scenario, one can imagine several chargers that are located around the work area. The reason for having more than one charger in this simulation is to be able to simulate the ability to navigating the robot to the closest charger when needed. Chargers can easily be added or taken away as needed for the simulation. The rest of the objects in the world represent different obstacles that the robots must navigate around.

As this project focuses on high level planning, the simplifications in this world representation is sufficient for the tests conducted in this thesis. These tests make the assumption that actions like inspecting, docking/undocking and charging are handled by the robot's lower level control, hence making it sufficient to simulate the success or failure of the action.



**(a)** The world map created in Gazebo. The green squares represent charging stations, the large orange cones represent objects of interest. The rest of the objects are obstacles that the robot must navigate around.



**(b)** The occupancy grid map created in the TurtleBot3 simulator by the built-in SLAM functionality. The white area represent collision free areas, black represent occupied and inaccessible areas, and gray represent unknown areas where the LIDAR was not able to reach.

**Figure 4.4:** 3D world map in gazebo and 2D occupancy grid map for the simulated environment used for simulation testing.

The navigation package that comes with the TurtleBot3 needs a 2d occupancy grid map of the environment it is working in. This map is created using the built-in SLAM feature. When running the SLAM functionality, the robot is able to move around in the world while constructing a map of the surroundings using its LIDAR and the robot's pose. The constructed world map can then be stored as a *pgm* and *yaml* file to be used by the navigation stack. The resulting occupancy grid map of the world in Figure 4.4a can be seen in Figure 4.4b. Here, the white area represent known, collision free areas, black represent occupied and inaccessible areas, and gray represent unknown areas where the LIDAR was not able to reach.

## 4.7 Lab experimental test setup

Both in the specialization project [4] and this thesis, it has been stated that one of the advantages with the design of the system presented is that it is easy to integrate into a physical robotic platform due to the use of ROS. In order to evaluate this claim, and to test the robustness of the system when adding physical disturbances, the system was implemented and tested using a TurtleBot3 and a simplified environment setup.

The TurtleBot3 comes as a building kit where all parts of the robot, both hardware and software, has to be set up. The building instructions for assembling the robot were very straight forward, as was the software setup, involving installation of a custom Ubuntu image to the Raspberry Pi Single Board Computer (SBC). As the ROSPlan framework officially supports ROS Melodic, the Raspberry Pi was also set up using the ROS Melodic image provided by Robotis, the producers of the TurtleBot3. However, due to problems with USB peripherals and network setup on the Raspberry Pi, the ROS Noetic image was tested instead, solving the problems. Running Noetic

on the TurtleBot3 while the remote computer with the ROSPlan system is running Melodic is not ideal, as it could cause problems in the communication between the computers due to discrepancies between the versions. As most of the computation is taking place on the remote computer, with the communication between computers consisting of sensor data and velocity commands, this should however not be a big problem due to the ROS messages used being standard messages that are unchanged between the versions.

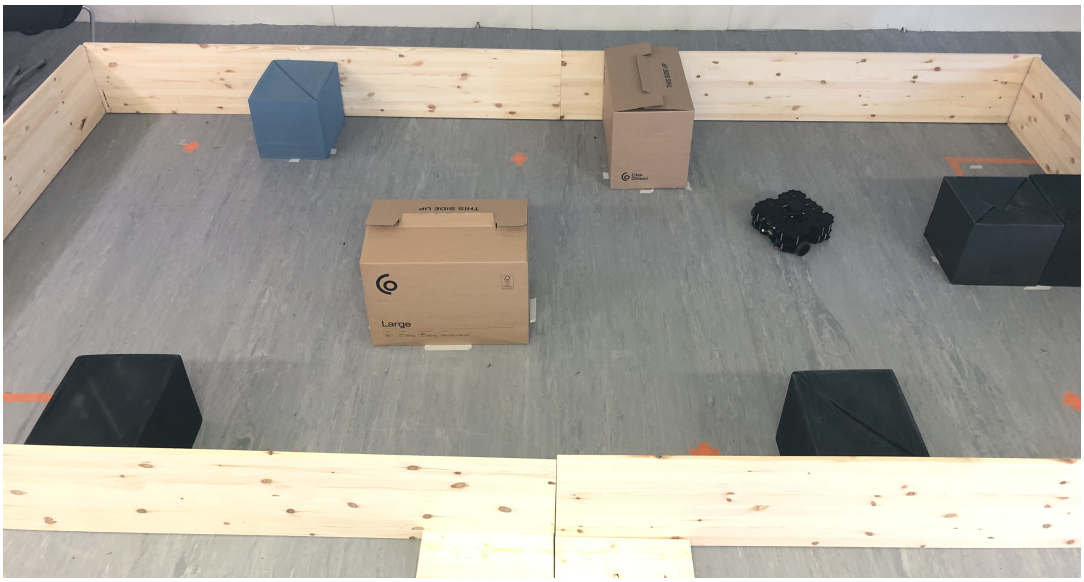



**Figure 4.5:** Overhead view of the robot test area with the TurtleBot3 robotic platform. Walls and boxes are treated as obstacles, while orange markers on the floor illustrates waypoint locations. Charging areas are illustrated as orange squares on the floor and can be spotted in the bottom left and top right corners.

To perform test missions, a simple environment was set up using walls and boxes as obstacles. Waypoints and charging areas were marked using orange tape on the floor. As the TurtleBot3 does not come with a charging station, this was only simulated with a square on the floor. The test area was designed with passages of varying width in order to make navigation between waypoints more challenging. The total area of the enclosure was significantly smaller than the simulated environment described in section 4.6.3, mainly due to constraints in the room used for the experiments. Figure 4.5 shows an overview of the test area described.

**Figure 4.6:** Screenshot from Rviz showing the occupancy grid map for the test area with the global costmap (colored overlay) and waypoints (white squares).

The ROS navigation stack needed an occupancy grid map of the test area in order for the path planning algorithm to work. This was created using the Simultaneous Localization And Mapping (SLAM) feature described in section

4.6.3. The resulting occupancy grid map can be seen in Figure 4.6. In order to make the robot behave in a desired manner when moving in the environment, the ROS navigation stack that is used by the TurtleBot3 had to be tuned. This involved tuning parameters of both the global and local path planners to ensure smooth paths between waypoints, without the robot going too close to obstacles. Tuning the ROS navigation stack is covered thoroughly by Zheng [48], however as this involves lower level planning that is generally outside the scope of this project, it is not described further here.

# 5

# Results and discussion

In this chapter, results of the work described earlier in this thesis are presented. The results are discussed in an effort to answer the research questions for this thesis. The first part of this chapter presents results and discussion of the PDDL design described in chapter 3. This part will be followed by the results and discussion of the ROSPlan functionality extension described in chapter 4. Lastly, the results of the lab tests described in section 4.7 are presented and discussed.

## 5.1 Domain and problem design

This section presents an evaluation and discussion of the PDDL design described in chapter 3. The domain's performance on an Inspection and Maintenance (I&M) mission is tested, and the pros and cons of the PDDL design is discussed.

### 5.1.1 PDDL design evaluation

The purpose of the PDDL design test was to evaluate how the PDDL domain and problem design performed for an industrial I&M mission. In order to ensure that the design worked with different planners, using different planning approaches, and to decide which of the planners to use for the rest of the evaluation tests in this project, the test was performed running both the POPF and LPG-td planner with the same domain and problem file.

The test was conducted by providing the AI planner with the PDDL domain and problem described in chapter 3. As the LPG-td planner is a stochastic anytime planner that does not produce the same plan every time it is run on the same problem, the planner was run three times and all three plans presented below. The POPF planner is a deterministic planner, meaning that the planner will output the same plan every time, when run on the same planning problem. This meant that the POPF only had to be run a single time for this test.

The POPF planner is simple to run and only requires three arguments:

```
$ ./popf domain.pddl problem.pddl solutionfilename
```

The plan is then output in the solution file. The LPG-td planner is a bit more complicated to run as the number of iterations and maximum CPU-time needs to be tuned in order to get the best plan possible. For this test the planner was run with the command

```
$ ./lpg-td -o domain.pddl -f problem.pddl -n 20 -cputime 60 -out
  solutionfilename
```

Where the arguments are:

- -o: domain file name
- -f: problem file name
- -n: max number of solution iterations
- -cputime: maximum CPU-time (in seconds)
- -out: solution file name

As the problem size of this mission was relatively small, large deviations in the results of the planners were not expected. The focus of this test was mainly to evaluate the performance of the domain and problem design using planners with different planning algorithms, as well as testing speed, ease of use and accuracy of the planners to decide what planner to use for the rest of the tests in this project.

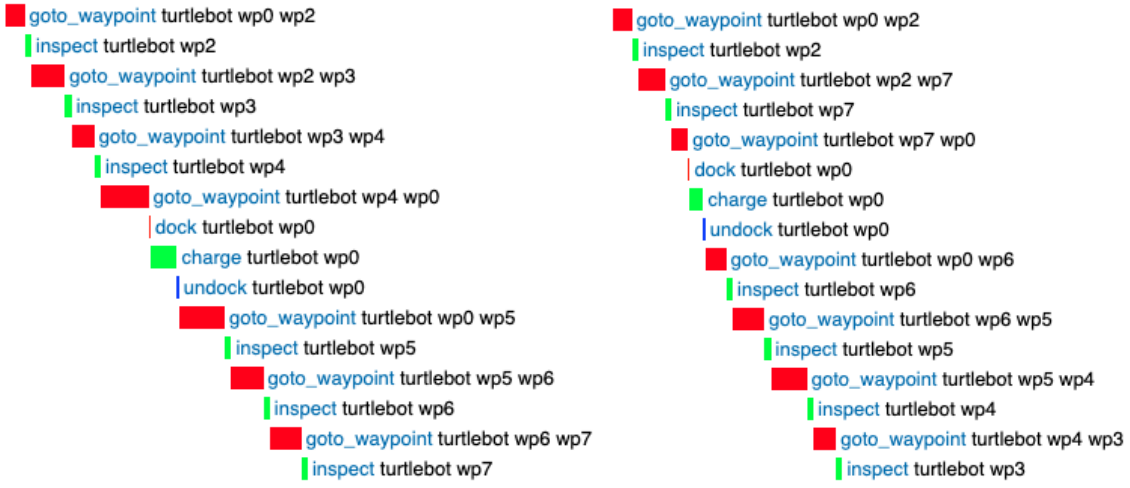
### 5.1.2 PDDL design results

Running both AI planners, four plans were obtained. One from the POPF planner and three different plans from LPG-td. Table 5.1 shows a comparison of different metrics between the four plans. All plans contained the same number of actions, but ordered in four different way, giving different resulting performance. The results clearly shows that the POPF planner produced a significantly less efficient plan than any of the LPG-td plans, with a plan run time of over 100 seconds more than the best plan. This is likely due to LPG-td being a stochastic planner, designed to explore the solution space more broadly. POPF, on the other hand, being a deterministic planner, can often find a satisfactory solution more quickly, though often with lower quality. This was very apparent when running the two planners, and can be seen in the last row of table 5.1. While POPF gave its result in just 20 milliseconds, LPG-td spent between 3 and 7 seconds on improving its initial plans and finding the best plan possible. Though 7 seconds is not a very long time compared to the over 350 seconds of the plan run time, it is a significant time spent on solving a very small planning problem.

| | POPF | LPG-td 1 | LPG-td 2 | LPG-td 3 |
|---|---|---|---|---|
| Number of actions | 16 | 16 | 16 | 16 |
| Plan run time [s] | 463 | 368 | 355 | 370 |
| Distance traveled [m] | 35.7 | 26.3 | 26.9 | 27.5 |
| Total battery used [%] | 152 | 115 | 117 | 119 |
| CPU-time [s] | 0.02 | 3.74 | 6.85 | 7.37 |

**Table 5.1:** Metrics for the plans produced by POPF and LPG-td. The first four rows contain metrics from the plan. The last row, CPU-time, describe the time spent by the planner to produce the plan.

Figure 5.1 shows a comparison between the Gantt charts of the POPF plan and the LPG-td plan 2, the best out of the three plans produced by LPG-td. Here it can be seen that the choice of going from waypoint 4 to waypoint 0 and back to waypoint 5 in order to charge the batteries adds a lot of time to the POPF plan. This transit time is lower in the LPG-td plan 2, due to the distance between waypoint 7 and the charging station at waypoint 0 being closer. This also affects the time spent at the charging station, as charging time is determined by the battery charge level at the start of the action.




**(a)** POPF plan **(b)** LPG-td plan 2

**Figure 5.1:** Comparison of the Gantt charts of the POPF plan and the best out of the three plans produced by LPG-td

Figures 5.2 and 5.3 shows how battery level and distance traveled changes over time for the four different plans. These plots give the same impression as table 5.1 of LPG-td producing significantly higher quality plans than POPF, with the POPF plan traveling significantly longer and spending more battery than the three other plans.

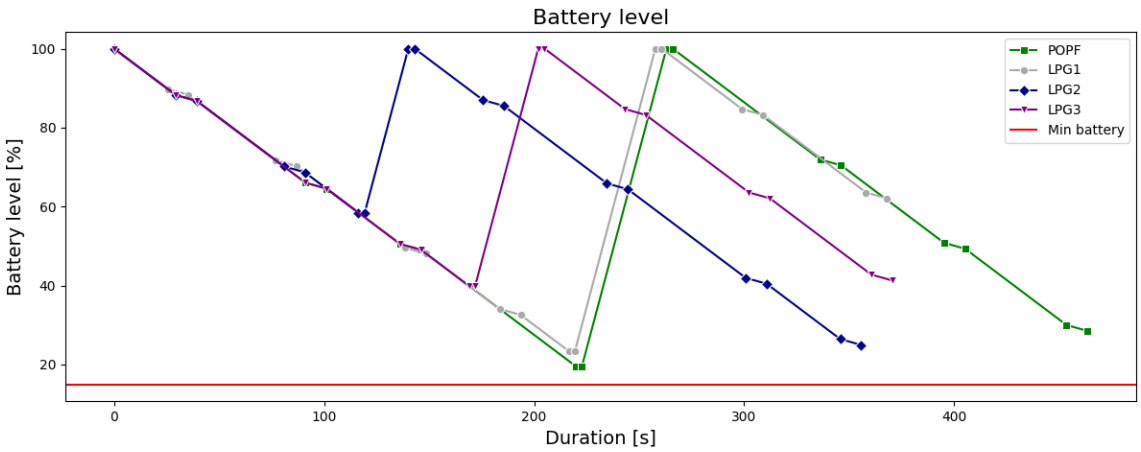



**Figure 5.2:** Battery level of the robot related to time. Notice the linear characteristics of the discharge rate, due to the simplified battery model in the PDDL domain. This is not an accurate representation of a battery system.

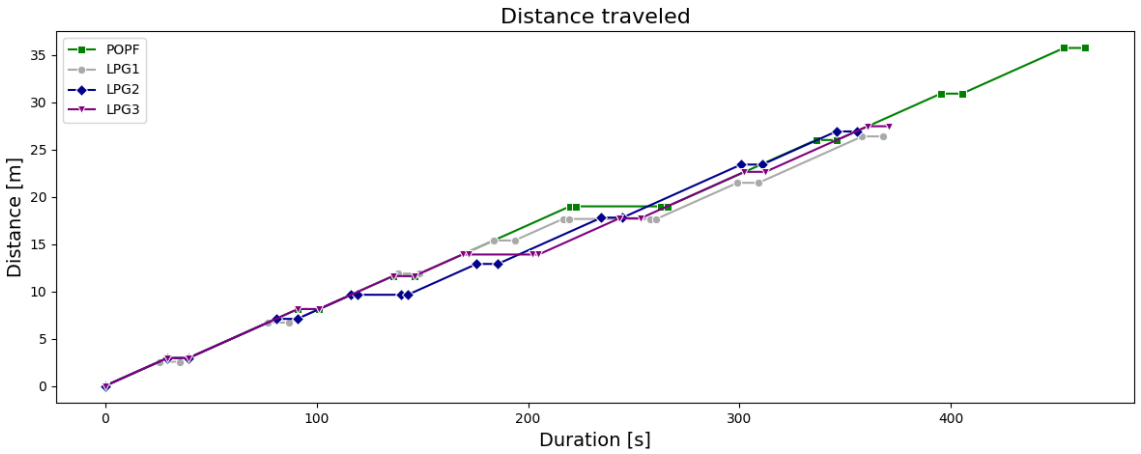



**Figure 5.3:** Distance traveled by the robot related to time.

The PDDL design test shows that the domain and problem design presented in this project is able to produce a functional planning problem that is able to produce a sequence of actions for an I&M mission. The test shows that at least two different planners, with different planning approaches, are able to use the domain and problem to produce plans. The domain allows battery to be simulated in the planning problem and direct the robot to a planner when needed, in order to not deplete the battery below the minimum charge specified. The design allows the planner to use the total traveling distance as a minimization variable to produce the best plan possible. Due to the generalized nature of the domain description, different missions can be defined in different environments, with varying amounts of waypoints and several robots. This makes the design scalable and adaptable.

One inherent problem with the domain design is very obvious when looking at Figure 5.2, and is the unrealistic nature of the robot battery simulation. As the battery is simply simulated as decreasing or increasing by a linear value when moving or charging, the model does not take into account the nonlinear nature of a battery and discharge rate. Not only is a battery nonlinear, but the discharge rate will also change based on the environment the robot is moving in, such as inclination, terrain, and variations in acceleration and speed. Energy is also spent on other

processes and components on the robot, causing the battery level to decrease even when the robot is not moving. in this project, this simplified modelling of battery depletion was chosen to reduce complexity. However, it is obvious that this model quickly will fall apart when deployed on a real robot, and a different way of modelling the battery will be needed.

One solution to the battery modeling problem is to use a custom interface that reports the real battery value to the Knowledge Base. With this feature, the battery model in the PDDL domain does not have to be completely accurate. It would simply serve as an initial guess for when the battery would need to be charged. The plan could then occasionally be re-evaluated with the updated battery level. This would also allow the system to react to unexpected events related to battery level, such as unexpected high battery depletion caused by a faulty battery cell.

The results of the comparison between planners showed a significant difference between the two planners, even for such a small planning problem. The LPG-td planner produced significantly higher quality plans than POPF, however this was at the cost of processing time. The POPF planner produced valid plans of lower quality than LPG-td, but at a fraction of the processing time. Another benefit of the POPF planner in regard to this project is the ability to produce the same plan for the same planning problem every time. This is a big advantage when running performance tests on the ROSPlan system in order to ensure reproducibility of the tests. These factors formed the basis for the decision of using the POPF planner for the rest of the tests in this thesis.

## 5.2 ROSPlan functionality extension

In this section, the results of the functionality extensions are presented and discussed. A discussion of the system as a whole is provided at the end of the section.

The ROSPlan feature extensions designed in this project were tested in an effort to evaluate and verify its functionality in a simulated mission environment. All tests were run in the Gazebo simulator presented in section 4.6 and using the PDDL domain description and problem instance presented in chapter 3. The mission consisted of eight waypoints, with there being six objects of interest and two battery charging stations. Figure 5.4 shows a 2D map of the Gazebo world model used for the simulated missions. In this figure, the locations of the waypoints corresponding to the mission waypoints are illustrated with white squares and the waypoint number. Waypoint 0 and 1 (wp0 and


**Figure 5.4:** 2D representation of the test environment with waypoint placement visualized in white.

wp1) represent the location of charging stations, while the rest of the waypoints represent the location from where the robot should inspect objects of interest.

Three different tests were run in order to test all functionality extensions implemented in this project. Section 5.2.1 describes testing and evaluating the operator interaction functionality, as well as the re-planning feature. Section 5.2.2 describes the testing and evaluation of the goal removal functionality for situations where a goal was not obtainable, even when attempted several times, while section 5.2.3 describes the results of the feature for handling blocked paths and the robot recovering from loosing its known position in the ROSPlan Knowledge Base.

### 5.2.1 Operator interaction and re-planning

The operator interaction and re-planning test was conducted in order to evaluate the functionality described in sections 4.4 and 4.5. The purpose of the re-planning feature is to detect when an action has failed and the robot needs to run a new planning sequence. The action failing could be caused by problems in the environment, due to a blocked path, or an operator triggered event, like an inspection photo being rejected. This feature is a vital part of a robotic AI planning system, as it allows the system to handle unforeseen obstacles during plan dispatch and adapt to the situation as it occurs.

For this test, all action failures were simulated through the operator interaction feature. This ensured reproducibility as it provided control over which actions failed, and at what time. It also made it possible to test the functionality of the operator interaction feature. The re-planning feature was tested by triggering an inspection action failure at waypoint 3 and 6, necessitating re-planning in order to add the failed goals back into the mission plan. The test was run on the same mission as the PDDL design evaluation tests, with 6 objects to be inspected, two chargers and a minimum battery requirement of 15% capacity. The POPF planner was used for the mission, mainly do to the low CPU-time compared to LPG-td, reducing the mission execution time, and its ability to reproduce the same plan every time, making it possible to run the test several times. Running the mission as described, an initial plan was generated and dispatched. This plan can be seen in Code 5.1 and is a complete plan for the I&M mission. The plan included inspection of all objects of interest, as well as charging at the waypoint 0 charger.

```
0.000: (goto_waypoint turtlebot wp0 wp2)  [29.411]
29.412: (inspect turtlebot wp2)  [10.000]
39.413: (goto_waypoint turtlebot wp2 wp3)  [51.740]
91.154: (inspect turtlebot wp3)  [10.000]
101.155: (goto_waypoint turtlebot wp3 wp4)  [34.986]
136.141: (inspect turtlebot wp4)  [10.000]
146.142: (goto_waypoint turtlebot wp4 wp5)  [56.045]
202.188: (inspect turtlebot wp5)  [10.000]
212.189: (goto_waypoint turtlebot wp5 wp0)  [70.178]
282.367: (dock turtlebot wp0)  [1.000]
283.367: (charge turtlebot wp0)  [42.354]
325.721: (undock turtlebot wp0)  [1.000]
326.721: (goto_waypoint turtlebot wp0 wp6)  [32.558]
359.280: (inspect turtlebot wp6)  [10.000]
369.281: (goto_waypoint turtlebot wp6 wp7)  [48.270]
417.552: (inspect turtlebot wp7)  [10.000]
```

**Code 5.1:** Initial plan for operator interaction and re-planning test.

After the inspection action at waypoint 3 was performed (line 4 in Code 5.1), the operator rejected the photograph while the next action (*goto_waypoint turtlebot wp3 wp4*) was dispatched. This meant that the robot was located between waypoints 3 and 4 at the moment that the photograph was rejected. In order to keep the flow of the mission and not causing the robot to stop abruptly between waypoints, the dispatch was then continued until the robot reached waypoint 4 before being canceled. As described in chapter 4, the system then reset the *photographed wp3* proposition to false and re-added it as a goal in the problem instance before reinitiating the planning sequence. The terminal output from the operator interaction node can be seen in Code 5.2.

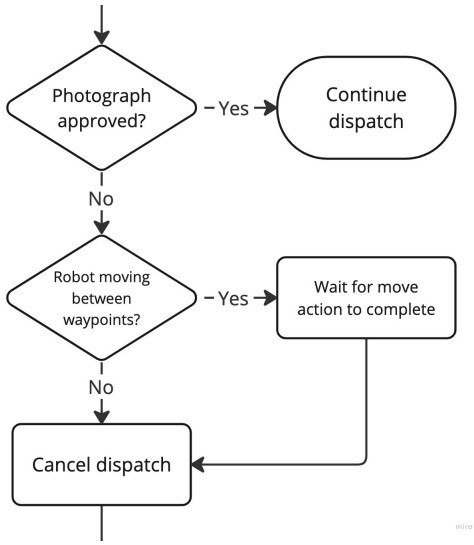
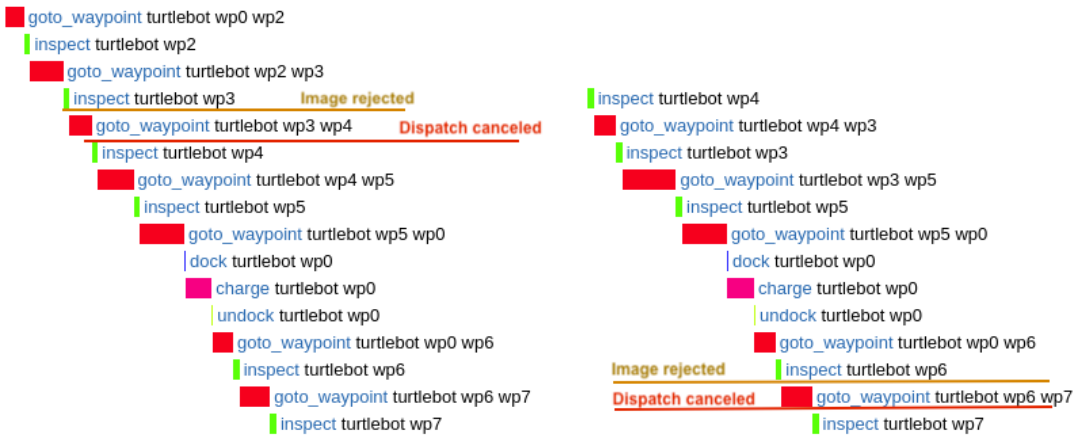



**Figure 5.5:** Closeup of operator interaction node flow chart, illustrating how it handles a photo rejection.

```
Photo taken of wp3. Press ENTER to reject or wait to approve photo.
Photo rejected. Re-adding wp3 as goal and replanning.
Dispatch canceled. Replanning.
```

**Code 5.2:** Output from the operator interaction node after the first three inspection actions.

When the dispatch is canceled and the re-planning is initialized, the planning loop described in Section 4.4 starts a new iteration. This means the problem interface creating a new problem desecration based on the information in the Knowledge Base. As the failed action is added back as a goal when the inspection photo is rejected, the goal will also be part of the next plan. This can be seen in Figure 5.6b. The loop continues by running the planner interface, parsing interface and plan dispatch, making the robot continue the mission from where it left off.




**(a)** Initial plan. **(b)** Second plan.

**Figure 5.6:** Planner output showing the initial and second plan of the operator interaction and re-planning test, including image rejection and dispatch cancelation. Note that the *inspect turtlebot wp3* goal has been re-added to the Knowledge Base after photo rejection and is also part of the second plan.

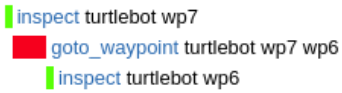



**Figure 5.7:** Third and final plan for the operator intervention and re-planning evaluation mission, generated after the inspection of waypoint 6 was rejected by the operator.

Continuing the mission, the robot followed the plan in Figure 5.6b. When reaching waypoint 6 and inspecting the object of interest, the operator once again rejects the photo. This causes the same reaction as described above, adding the goal back into the Knowledge Base and canceling the dispatch when having reached waypoint 7. The planning loop was run a third iteration, generating the plan in figure 5.7. At this point, the mission goals were (*photographed wp6* and *photographed wp7*). The rest of the dispatch mission was then run without failures, successfully completing the dispatch after inspecting waypoint 6 for a second time.

The results of this evaluation test illustrate the functionality of the planning/re-planning loop created for this project. Through the planning node, the system is able to launch the planning sequence that plans and dispatch the mission, as well as handle failed actions and redo the planning sequence when necessary. This functionality that is crucial for an autonomous robotic system using AI planning, as it allows the system to handle a dynamic environment where unplanned events may occur, and as this functionality is not part of ROSPlan by default, it is an important addition to the framework. It is worth noting that this feature is needed mostly because of the ROSPlan framework using an offline planning approach, and that a system with an online planning approach would not necessarily be in need of such a feature, due to the nature of the online approach being able to adapt to changes in the environment by itself.

The test also proves the functionality of the operator interaction feature, adding an operator-in-the-loop feature to the planning system. One can imagine that this feature could be expanded to involve more actions and ways of taking control of the robot, like manually adding and removing goals, or manually overriding the dispatch for operational or safety reasons. For this project, the feature is implemented as a ROS node that is controlled and outputs to a terminal window, but this could easily be integrated into some form of Graphical User Interface (GUI). As one of the goals

of automating robotic missions is to reduce the need for human interaction during the mission, one can also imagine a system where the verification of the inspection photos are done by a non-human operator, using image processing to approve and reject the photos. This would eliminate the need for an operator verifying the photos in real-time while the robot is dispatched.

As described earlier, the plan dispatch will continue if the photo is rejected during a movement between waypoints, and will only be canceled once the robot has reached the end waypoint. This is a design decision that was taken to keep the flow of the mission and giving a more predictable behavior for people interacting with the robot. It was also done to avoid the problem of an unknown robot position due to the movement action being canceled, as described in section 4.4.1. However, this functionality could be changed if desired, for example making the robot go straight back to the previous waypoint or stop for re-planning immediately. What behavior is best for the robot and mission environment it is operating in will often vary and would need to be adapted to the specific circumstances of the mission.

### 5.2.2 Goal removal

The goal removal feature was tested in an effort to evaluate the functionality described in section 4.4.3, removing goals when the same action has failed repeatedly. This feature is important as it allows a mission to continue even though one of the goals are not achievable, increasing the robustness of the autonomous I&M mission. To test this feature, the same mission described earlier was conducted using the operator interaction feature to simulate failed actions.

During the mission, the operator decides to reject the photograph at waypoint 3. In a real world mission, this rejection could for example be due to a photo where the object of interest is out of focus or not visible. This should cause a dispatch cancellation and re-planning, with the *photographed wp3* proposition added back as a goal. When the robot returns to waypoint 3, a new photo is taken and once again rejected by the operator. When the action has failed 3 times, a number that can be changed as desired, the *photograph wp3* goal should be removed to allow the mission to continue without interruption.

Running the mission problem instance as described, the same initial plan, Code 5.1, as in the operator interaction and re-planning test was obtained. The plan was dispatched, and the robot started the inspection mission. After having tried and failed three times to inspect waypoint 3, the goal removal functionality was activated. Code 5.3 shows the terminal output of the operator interaction node after inspection of waypoint 3 has failed for the third time. This output shows that the goal was removed after the third failure and replanning initiated without *(photographed wp3)* as a goal.

Table 5.2 shows the sequence of actions for the goal removal mission, with the *goto_waypoint* actions excluded for clarity. Here it can be seen that the robot returns to waypoint 3 after failure in order to re-attempt inspection. After the third failed attempt, the goal is removed, and the mission continues without re-attempting to inspect waypoint 3. Note also that the waypoint at which the charging action is performed has been changed from the original plan, as this reduced the distance traveled after re-planning.

```
Photo taken of wp3. Press ENTER to reject or wait to approve photo.
Photo rejected. Re-adding wp3 as goal and replanning.
[WARN] Goal ('photographed', 'wp3') has failed 3 times. Removing goal from the
    Knowledge Base.
Dispatch canceled. Replanning.

Photo taken of wp6. Press ENTER to reject or wait to approve photo.
Photo approved, continuing mission.
```

**Code 5.3:** Terminal output from the operator interaction node showing goal removal after 3 failed attempts

| | Action | Comment |
|---|---|---|
| 1 | Inspect wp2 | Success |
| 2 | Inspect wp3 | Failed 1st |
| 3 | Inspect wp4 | Success |
| 4 | Inspect wp3 | Failed 2nd |
| 5 | Inspect wp5 | Success |
| 6 | Charge wp1 | Success |
| 7 | Inspect wp3 | Failed 3rd. Goal removed |
| 8 | Inspect wp6 | Success |
| 9 | Inspect wp7 | Dispatch successfully finished |

**Table 5.2:** Goal removal test mission, excluding *goto_waypoint* actions for clarity.

The results of this test show the goal removal feature working as desired, removing repeatedly failing goals from the Knowledge Base. This feature builds on the re-planning functionality to form the basis of an autonomous robotic system that is able to handle changes to the initial plan and adapt to a dynamic environment. In this evaluation test, the failure threshold was set to three failures before goal removal. This seems to be a reasonable number to ensure that the goal really is unachievable, while not spending unnecessary time on going back to the waypoint. However, as mentioned earlier, this threshold can be changed as desired.

### 5.2.3 Blocked path and unknown position recovery

To ensure that the system is able to handle events where the path to a waypoint is blocked, a test was set up using the Gazebo simulator and the same mission described earlier. During simulation in Gazebo, it is possible to move and add objects, making it possible to change the environment the robot is operating in. This is similar to how an environment can change in the real world, and handling these changes is an important aspect of autonomous robot operations in such environments.

During the mission simulation, a new object is placed on the map, blocking waypoint 5. This makes the robot unable to reach the waypoint. The expected result of this test was for the *goto_waypoint wp5* action to fail due to the navigation feature of the TurtleBot3 not being able to find a valid path. This would then trigger a re-planning by the re-planning feature described earlier. As the action fails between two waypoints, the position of the robot is now unknown in the Knowledge Base. This problem is described in section 4.4.1 and is caused by the fact that when

an action fails, the end effect of the action is not applied. To solve this problem, a new waypoint should be added at the position of the robot. A re-planning should then be initialized.

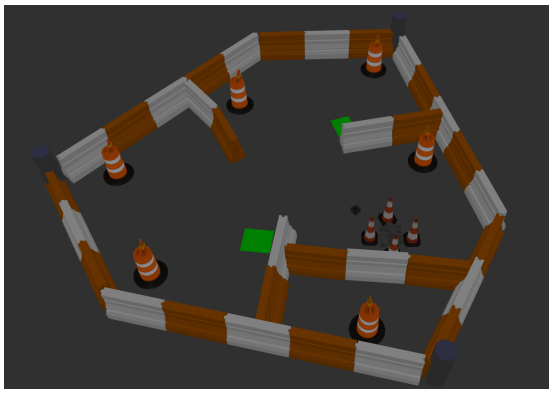
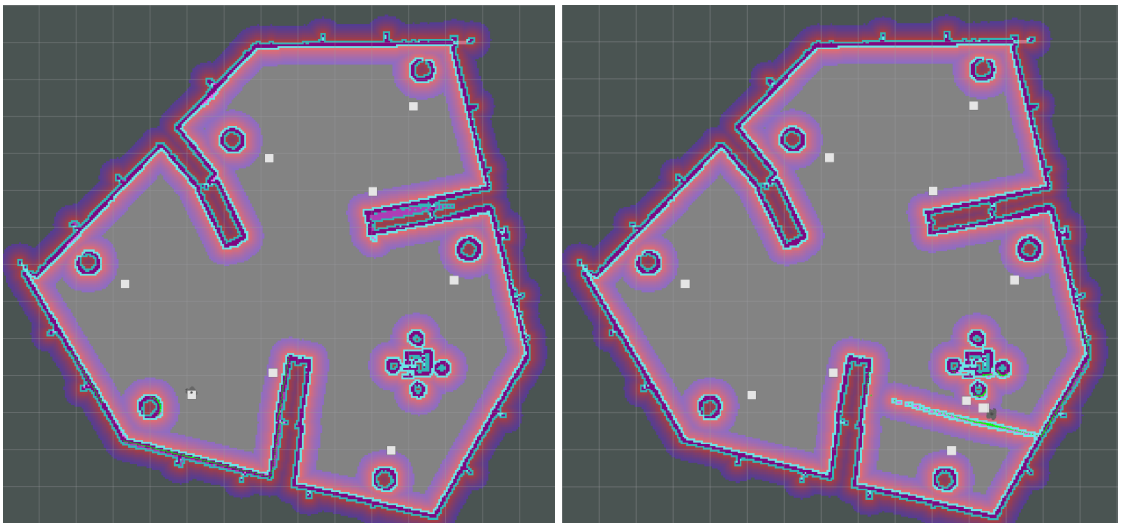


**Figure 5.8:** Figure showing waypoint 5 (lower right corner) being blocked and unreachable by the TurtleBot3.

To ensure the robustness of the feature, two different scenarios were tested. For the first scenario, the obstruction was removed after a second action failure, and the robot was therefore expected to finish the mission dispatch successfully, including a successful inspection of waypoint 5. In a real world environment, this scenario could represent a blocked path due to a moving object or person being in the way of the robot for some time before moving away from the path. The second scenario involved the object not being removed, hence necessitating the goal to be removed with the goal removal feature. This scenario simulated a path being permanently blocked, for example due to a closed door or an object laying across the robot's path.

Running the same mission as the other evaluation tests, the robot was dispatched and performed all tasks until reaching waypoint 5. Upon approaching waypoint 5, the onboard LIDAR on the robot detected the wall placed in front of the waypoint. This can be seen in Figure 5.9. As this wall was not part of the occupancy grid map that the TurtleBot uses for navigation, it was not considered before it was detected by the LIDAR. The TurtleBot3 navigation stack is able to do obstacle avoidance and path re-planning, and the robot therefore then tried to find a new route to the waypoint. As there was no alternative route, the TurtleBot eventually gave up and returned feedback to the action interface, causing a failed action. Code 5.4 shows the terminal output of the planning node after the action have failed.



**(a)** Before reaching the wall in front of wp5 **(b)** After attempting to reach wp5

**Figure 5.9:** 2D map showing before and after the robot has attempted to reach waypoint 5. Note the wall detected by the robot's LIDAR and the new waypoints (white squares) generated in order to recover from the unknown position state

As the action failed after leaving the previous waypoint, but before reaching waypoint 5, the position of the robot in the Knowledge Base (proposition (*robot_at*)) was now unknown. Because of this, the planning node entered a special case, as seen from Figure 5.10. In this special case, the node adds a new waypoint at the location of the robot in order for the Knowledge Base to regain the knowledge of where the robot is and for the Problem Interface to create a valid planning problem.

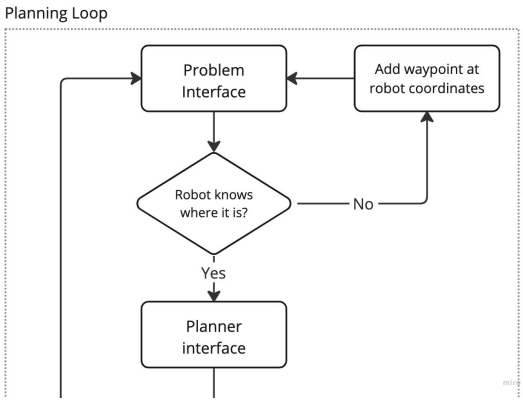



**Figure 5.10:** Closeup of the planning node flow chart, showing how the node handles an unknown robot position.

Both the first and second test showed good results and were able to re-plan the mission without the planner failing, proving that the method of adding waypoints worked. The first test illustrated that the system was able to handle a temporarily blocked path and recover from an unknown position when the *goto_waypoint* action failed. It also showed that the robot was able to recover and continue the mission when the wall was removed, making the waypoint reachable again. The second test illustrated that the robot was able to recover, even though the wall was not removed, using the goal removal feature that was tested earlier. This shows that the blocked path and unknown position recovery feature increases the robustness of the system by making it able to handle unforeseen changes to the environment the robot is operating in, a crucial aspect of a robotic mission in dynamic environments.

```
1 [ERROR]: Action failed, canceling dispatch
2 [INFO]: Goal has failed 2 times
3 [ERROR]: Dispatch was canceled without all goals being achived. Replanning
4 [INFO]: Calling problem generator.
5 [INFO]: Robot does not know where it is! Adding a new waypoint at its current
     position.
6 [INFO]: Calling problem generator.
```

**Code 5.4:** Terminal output from the planning node showing the robot recovering from an unknown position state.

One drawback of the functionality of the blocked path feature that was not seen by the simulation test, but that could pose a problem for a larger I&M mission, is the way the system handles removing a goal after failing to reach a waypoint. As the system only removes the goal from the Knowledge Base, not the waypoint itself, the unreachable waypoint is still included in the planning problem. In theory, this means that the waypoint could still be part of the new plan as an intermediate point between two other waypoints, even though it is still unreachable. For the domain used in this project, this is not a problem as intermediate waypoints are not needed. However, for other domains where intermediate waypoints are needed, for example a robot where the path planner has limited range, this could pose a problem. A reasonable solution for this issue could be to also remove the waypoint from the planning problem, making the planner unaware of the existence of this waypoint and not including it in its plans. It could also be possible to remove the links between the blocked waypoint and all other waypoint in the planning problem. This would have a similar effect to the other solution, making the planner unable to include an action

for the robot to go to the blocked waypoint. These approaches were not pursued in this project, mainly due to time constraints and the fact that the issue did not occur in testing.

Another issue that was not experienced during testing, but could cause issues with a larger planning problem is the introduction of new waypoints. For every waypoint that is added, the planning problem will grow, increasing the computational complexity. As the planning problem in this project is small and the AI planner used is efficient, this did not prove a problem in the testing described. However, one could imagine a large planning problem with many waypoints and goals, and where several waypoints prove to be blocked. This would add more complexity to an already large planning problem, eventually making the planning process slow and inefficient. One solution to this problem could be to remove the newly added waypoint after replanning. This would help keep the planning problem as small as possible, reducing the computational complexity.

The unknown position recovery feature designed for this project is a feature created to handle an inherent issue with the way PDDL is designed. As the effects of an action can only be applied *at start* and *at end*, problems will occur when an action is canceled before completion, as *at end* effects will not be applied. This is not only a problem for the position of the robot that is handled through the feature describe above, but also other effects, for example battery changes. During the simulation tests, this problem became apparent as the battery charge level would not be reduced when a move action failed, regardless of the distance the robot had covered. Thus, the unknown position recovery only solves one of the problems caused by the end effects of an action not being applied, and more systems would need to be developed to handle the other effects.

## 5.3 Lab experimental test results

In order to test how the physical system performed compared to the simulations, the same tests as described above were run on the lab test setup. In addition to these tests, a full mission was performed with all features created for this project. This was done in order to evaluate how the system responded when several events were triggered in the same run. The final test run was performed with the following events:

- Inspection of waypoint 2 fails on first try
- Waypoint 3 is permanently blocked
- Waypoint 5 is temporarily blocked, but then cleared

### 5.3.1 Setup

Setting up the test environment was straight forward and only involved placing objects and obstructions that the robot could interact with. Building and setting up the TurtleBot3 was also fairly easy, with everything needed for the process being provided in the setup tutorial. The test environment was mapped using the SLAM functionality that is included with the TurtleBot3 in order to create an occupancy grid map. This map is used by the navigation stack in order to provide path planning capabilities.

Setting up and tuning the navigation stack was the first task that provided some challenges. The simulated TurtleBot3 used in this project had a well tuned navigation stack that did not need any tuning out of the box. These tuning parameters were, however, not suited for the physical test environment. This was most likely due to the map being significantly smaller than the simulated environment, demanding tighter turns and closer approaches to the walls and objects. As the navigation stack is out of the scope of this project, it was only tuned to a sufficient degree to where the robot was able to traverse the area without major problems.

**Figure 5.11:** Occupancy grid map with waypoints illustrated as white squares. Upper left and lower right waypoints are chargers, with the rest being objects of interest

As the area was smaller than the simulated environment, the number of waypoints placed in the area was reduced from 8 to 7. Two chargers were placed on opposite sides and 5 inspection waypoints were dispersed throughout the area. The waypoints can be seen as white squares in Figure 5.11.

### 5.3.2 Experimental results

Both the individual functionality tests, identical to the previously described tests, and the full mission test went to plan without major issues or deviations from the simulated tests. This result was as expected due to the Gazebo simulator creating a very accurate simulation environment and ROSPlan being a modular and easily integrable framework, due to the use of ROS. The biggest problems experienced during the tests however were caused by the TurtleBot3 navigation stack. As not a lot of time was spent on fine-tuning the global and local planner to the test environment, some issues were experienced due to narrow spaces and waypoints close to the wall. These problems manifested themselves as the robot not being able to do sharp turns around obstacles and sometimes getting stuck in corners or close to the walls. This could eventually cause the navigation to fail, triggering an action failure

**Figure 5.12:** Test area with waypoint 3 and 5 blocked by wooden board. The narrow passage between the board and the cardboard box made path planning extra challenging.

in ROSPlan. As these problems only occurred occasionally, this was not seen as a problem for the results of the tests, rather showing the strengths of the system as it was able to recover, using the lost position recovery feature developed for this project.

The final, full mission run evaluation test was performed in order to test all functionality of the extended ROSPlan framework. As described in section 4.7, the test was set up with several events to test the different features of the system. For this test, the goal was to verify that all the features worked together and did not interfere with each other. Using a wooden board as an obstruction in front of waypoints 3 and 5, as seen from Figure 5.12, the test was initiated from the charging area at waypoint 0. The system preformed mostly as expected, only struggling with navigating between a few waypoints in narrow spaces. As the wooden board used as an obstruction created a very narrow corridor around waypoint 3, this created an extra challenge for the navigation stack trying to find a path to the blocked waypoint. Disregarding one failed action due to the poorly tuned navigation stack, the system was able to perform the mission as expected, handling all events occurring during dispatch. Table 5.3 shows the resulting sequence of actions, excluding *goto_waypoint* actions, for the full mission run.

The robot integration evaluation test proved that the claims of the ROSPlan framework being easily implemented on a physical system to be true. With about a day's work, the physical system was set up and integrated with the ROSPlan. No adaption to the code was needed in order for the system to function, only the parameters in the problem instance had to be change in order to tune the domain to the smaller test area. It is worth noting that this test was a best case situation in a controlled environment and with a robot that is tailor-made for use with a ROS system. A different robot could prove to be more difficult to integrate into the system, and testing this would be a natural next step in this project, if time allowed it.

| | Action | Comment |
|---|---|---|
| 1 | Inspect wp2 | Failed photograph |
| 2 | Inspect wp3 | Blocked x3. Goal removed |
| 3 | Inspect wp2 | Success (2nd attempt) |
| 4 | Inspect wp6 | Success |
| 6 | Charge wp1 | Success |
| 7 | Inspect wp5 | Blocked x1. Obstruction removed and success |
| 8 | Inspect wp4 | Success. Dispatch successfully finished |

**Table 5.3:** Resulting action sequence of the full mission run during the robot integration test, excluding *goto_waypoint* actions for clarity.

# 6

# Conclusions and further work

In this chapter, the work presented in this thesis is summarized and concluded, based on the results and discussions presented in chapter 5. Suggestions for further work are also presented in this chapter.

## 6.1 Conclusions

The goal of this project has been to investigate systems and techniques in order to answer the research question: "*How can AI Planning aid in autonomous robotic inspection and maintenance missions in industrial environments?*".

To answer this question, the project was broken down into three subtasks. The first subtask involved designing a PDDL domain description and problem instance that could describe a robotic I&M mission. The focus of the design was on making the domain adaptable to different robots, making as many parameters definable in the problem instance. The domain was also designed with battery simulation to allow the plans to include automatically directing the robot to the charger when needed. The PDDL design was tested on two different AI planners, based on different planning algorithms, to ensure that the design was robust and suited for different planners. Through the testing, the domain and problem proved to satisfy the design requirements, producing valid plans that could be dispatched to a robot. The design is scalable and allows for defining missions with different amounts of waypoints, charging stations and robots, using the same domain.

The second subtask involved implementing an AI planning system using ROSPlan and extending its functionality with features needed for an autonomous robotic system to performe Inspection and Maintenance (I&M) missions. The first and most important feature needed was functionality that allowed the system to re-plan the mission when needed in order to include new and updated information. This feature is crucial for the system to be able to handle unplanned events and operate in a dynamic environment. The feature was developed to run the planning sequence automatically when dispatch was canceled, avoiding the need for an operator to activate it. Through simulation and real-world testing, the feature was confirmed to work as intended and provided the basis for

the other feature extensions.

To enable operator-in-the-loop functionality in the system, a feature was developed to allow an operator to review the inspection photos taken by the robot and accept or reject the photo. This feature is a suggested first step in making an operator control system and more features, like adding/removing goal and dispatch override functionality, were suggested but not implemented. The possibility for automating the photo review using image processing was also discussed. Lastly, the image review feature was also used in order to test the re-planning and goal removal functionality.

In order to contribute to robustness of the system and allow dispatch to continue even with disturbances in the environment, the goal removal feature was proposed. The feature works by keeping track of failed actions and removing goals from the planning problem when an action has failed repeatedly. This ensures that the re-planning loop does not create a constant loop of trying to achieve an unobtainable goal. The feature was proved efficient through testing in simulator and lab testing, adding another layer of robustness to the system.

The last feature proposed in this thesis was developed to solve an inherent problem with how the PDDL domain handles movement. As an action needs to succeed in order for the *at end* effects to apply, the known position of the robot would be lost if a movement action failed for any reason. The lost position recovery feature enabled the system to regain knowledge of the robot position, enabling re-planning. This feature was also evaluated and proved through testing.

The third subtask involved testing and evaluating the systems developed in this project. All features of the system were tested and confirmed working through both simulation and real-world testing, focusing on how they were able to handle disturbances and unplanned events. The complete system was tested as a whole in a full mission run on a physical TurtleBot3, proving both the functionality of the system, and the ease of transferring the system from the simulator to a real robot.

In conclusion, we have found that the results in this thesis point toward an answer to the research question: AI planning can indeed aid autonomous robotic Inspection and Maintenance (I&M) mission by providing a tool for creating plans of action, as well as react to changes and unexpected events during plan dispatch. This helps to increase the autonomy of such missions by reducing the dependency on human operators. However, the planning problem of a robotic mission is only a small piece of the overall system, and AI planning in itself is no use without extensive supporting infrastructure around it. As the output of an AI planner in practice only consists of a sequence of action commands, the systems are needed, not only to carry out said commands, but also to handle unplanned events, update the planning problem, include an operator-in-the-loop, etc. The ROSPlan framework provides a starting point for such a system, providing tools for creating, and updating planning problems, creating plans with third-party AI planners, and dispatching the plans to an external system. In this thesis, new features has been added to this framework in order to provide the supporting infrastructure mentioned, extending its functionality and increasing its usability for autonomous robotic I&M missions.

## 6.2 Further work

The systems presented in this thesis were developed and tested on a TurtleBot3 robotic platform. This robot is a good prototyping and testing tool, but a natural next step for this project would be to test the systems described on a real I&M robot. This would involve creating interfacing systems that enable communication between the system and the robot. It would also include adapting the problem instance to a different robot.

For an autonomous robot to operate in an industrial environment, possibly including human interaction, safety and robustness is paramount. Developing safety systems and testing the framework in different environment and situations are therefor crucial before autonomous missions, like the ones described in this thesis, are deployed.

One feature that could increase both safety, robustness, and efficiency of the system is a better developed operator-in-the-loop system. With functionality that includes manual goal addition and removal, dispatch control, and safety override systems, the tool would allow an operator to take control of the mission dispatch when needed. The inspection photo feature could also be developed further and converted to a system controlled by AI and image processing.

Another feature that needs work in order for the system to be deployable on a real I&M robot is the battery modelling in the PDDL domain, as mentioned in section 5.1. Implementing functionality for feeding real battery information to the ROSPlan Knowledge Base, and using this information in the planning problem would create a more accurate representation of the battery, increasing the accuracy of the resulting plan.

# Bibliography


[1] SINTEF. "Autonomous robot missions with ai-based planning and acting (robplan)." (Oct. 2021), [Online]. Available: https://www.sintef.no/en/projects/2021/robplan/ (visited on 09/19/2022).

[2] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3, pp. 189–208, 1971.

[3] T. Estlin, D. Gaines, C. Chouinard, *et al.*, "Increased mars rover autonomy using ai planning, scheduling and execution," presented at the IEEE International Conference on Robotics and Automation, Rome, Italy, May 2007, pp. 4911–4918.

[4] J. Fillan, "Pddl plan validation system using rosplan and a turtlebot3 simulator," M.S. thesis, Department of Engineering Cybernetics, NTNU, Dec. 19, 2022.

[5] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting*. Cambridge, UK: Cambridge University Press, 2016.

[6] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th. Harlow, England: Pearson Education, Limited, 2021.

[7] D. Long and M. Fox, "Progress in ai planning research and applications," *The European Journal for the Informatics Professional*, vol. 3, no. 5, pp. 10–25, 2002.

[8] M. Ghallab, A. Howe, C. Knoblock, *et al.*, *Pddl - the planning domain definition language*, version 1.2, Oct. 1998.

[9] AIPLAN4EU Consortium. "Aiplan4eu." (2021), [Online]. Available: https://www.aiplan4eu-project.eu (visited on 05/23/2023).

[10] A. A. Transeth, I. Schjølberg, A. M. Lekkas, *et al.*, "Autonomous subsea intervention (seavention)," *IFAC-PapersOnLine*, vol. 55, no. 31, pp. 387–394, 2022.

[11] M. Fox and D. Long, "Pddl+ : Modelling continuous time-dependent effects," Apr. 2003.

[12] M. Fox and D. Long, "Pddl2.1: An extension to pddl for expressing temporal planning domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, 2003.

[13] A. Coles, A. Coles, M. Fox, and D. Long, "Forward-chaining partial-order planning," presented at the ICAPS 2010 - Proceedings of the 20th International Conference on Automated Planning and Scheduling, Toronto, Canada, 2010.

[14] A. Gerevini, A. Saetti, P. Toninelli, and I. Serina, "Lpg-td: A fully automated planner for pddl2.2 domains," presented at the 14th Int. Conference on Automated Planning and Scheduling, British Columbia, Canada, 2004.

[15] D. L. Poole and A. K. Mackworth, *Artificial Intelligence Foundations of Computational Agents, Foundations of Computational Agents*. Cambridge, UK: Cambridge University Press, 2017, p. 820.

[16] M. M. Veloso, A. Pérez, and J. G. Carbonell, "Nonlinear planning with parallel resource allocation," in *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, 1990.

[17] J. S. Penberthy and D. S. Weld, "UCPOP: A sound, complete, partial order planner for adl," presented at the International Conference on Principles of Knowledge Representation and Reasoning, 1992.

[18] A. Gerevini and I. Serina, "Lpg: A planner based on local search for planning graphs with action costs," presented at the Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, ser. AIPS'02, Toulouse, France: AAAI Press, 2002.

[19] M. Fox and D. Long, "The 3rd international planning competition: Results and analysis," *J. Artif. Intell. Res.*, vol. 20, pp. 1–59, 2003.

[20] A. Gerevini. "Experimental results for the test problems of the 3rd ipc." (Nov. 18, 2002), [Online]. Available: https://lpg.unibs.it/lpg/test-results/index.html (visited on 03/07/2023).

[21] ICAPS. "Icaps 2019 conference awards." (2019), [Online]. Available: https://icaps19.icaps-conference.org/awards.html (visited on 03/02/2023).

[22] ICAPS. "Ipc-04 results evaluation, and awards." (2004), [Online]. Available: https://ipc04.icaps-conference.org/deterministic/results.html (visited on 03/08/2023).

[23] M. Quigley, K. Conley, B. Gerkey, *et al.*, "Ros: An open-source robot operating system," presented at the ICRA Workshop on Open Source Software, vol. 3, Kobe, Japan, Jan. 2009.

[24] A. Ademovic. "An introduction to robot operating system: The ultimate robot application framework." (2022), [Online]. Available: https://www.toptal.com/robotics/introduction-to-robot-operating-system (visited on 10/03/2022).

[25] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, May 2022.

[26] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," presented at the International Conference on Intelligent Robots and Systems, Sendai, Japan, 2004.

[27] Open Source Robotics Foundation. "Gazebo, Robot simulation made easy." (2022), [Online]. Available: https://classic.gazebosim.org/ (visited on 12/14/2022).

[28] Planning at King's College London. "ROSPlan." (2022), [Online]. Available: https://kcl-planning.github.io/ROSPlan/ (visited on 11/10/2022).

[29] M. Cashmore, M. Fox, D. Long, *et al.*, "Rosplan: Planning in the robot operating system," presented at the Proceedings of the International Conference on Automated Planning and Scheduling, Jerusalem, Israel, Apr. 2015.

[30] M. Cashmore, M. Fox, D. Long, D. Magazzeni, and B. Ridder, "Opportunistic planning in autonomous underwater missions," *IEEE Transactions on Automation Science and Engineering*, vol. 15, no. 2, pp. 519–530, Jan. 2017.

[31] M. Cashmore, M. Fox, T. Larkworthy, D. Long, and D. Magazzeni, "Auv mission control via temporal planning," presented at the 2014 IEEE International Conference on Robotics and Automation, Hong Kong, China, 2014.

[32] L. Xue and A. M. Lekkas, "Comparison of ai planning frameworks for underwater intervention drones," presented at the Global Oceans 2020: Singapore – U.S. Gulf Coast, 2020.

[33] K. Rajan, F. Py, and J. Berreiro, "Towards deliberative control in marine robotics," in M. L. Seto, Ed. New York, NY: Springer New York, Dec. 2012, pp. 91–175.

[34] A. Popov, "Mission planning on the international space station program. concepts and systems," presented at the 2003 IEEE Aerospace Conference Proceedings, vol. 7, 2003, pp. 3427–3434.

[35] J. L. Bresina, A. K. Jónsson, P. H. Morris, and K. Rajan, "Activity planning for the mars exploration rovers," in *Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling*, ser. ICAPS'05, Monterey, California, USA: AAAI Press, 2005, pp. 40–49.

[36] S. Chien, R. Doyle, A. Davies, A. Jonsson, and R. Lorenz, "The Future of AI in Space," *IEEE Intelligent Systems*, vol. 21, no. 4, pp. 64–69, 2006.

[37] R. Steel, M. Niézette, A. Cesta, *et al.*, "Advanced planning and scheduling initiative MrSPOCK AIMS for XMAS," *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, vol. 1050, no. 3, 2009.

[38] V. Sanelli, M. Cashmore, D. Magazzeni, and L. Iocchi, "Short-term human-robot interaction through conditional planning and execution," presented at the Proceedings of the International Conference on Automated Planning and Scheduling, Pittsburgh, USA, Jun. 2017.

[39] B. Hoteit, A. Abdallah, A. Faour, I. A. Awada, A. Sorici, and A. M. Florea, "Ai planning and reasoning for a social assistive robot," presented at the 17th International Conference on Cognition and Exploratory Learning in Digital Age, Lisbon, Portugal, 2020.

[40] E. Karpas, S. Levine, P. Yu, and B. Williams, "Robust execution of plans for human-robot teams," presented at the Proceedings of the International Conference on Automated Planning and Scheduling, Jerusalem, Israel, Apr. 2015.

[41] K. Rajan and F. Py, "T-rex: Partitioned inference for auv mission control," *Further advances in unmanned marine vehicles*, pp. 171–199, 2012.

[42] Planning at King's College London. "ROSPlan." (2022), [Online]. Available: https://github.com/KCL-Planning/ROSPlan (visited on 11/07/2022).

[43] Planning at King's College London. "ROSPlan demos." (2022), [Online]. Available: https://github.com/KCL-Planning/rosplan_demos (visited on 11/10/2022).

[44] C. Muise, F. Pommerening, J. Seipp, and M. Katz, "Planutils: Bringing planning to the masses," in *32nd International Conference on Automated Planning and Scheduling, System Demonstrations and Exhibits*, 2022.

[45] J. Fillan. “Rosplan-inspection.” (), [Online]. Available: https://github.com/JFillan/ROSPlan-Inspection (visited on 10/12/2022).

[46] Open Source Robotics Foundation Inc. “What is a turtlebot?” (2022), [Online]. Available: https://www.turtlebot.com/ (visited on 10/03/2022).

[47] ROBOTIS. “Turtlebot3 simulations.” (2022), [Online]. Available: https://github.com/ROBOTIS-GIT/turtlebot3_simulations (visited on 11/07/2022).

[48] K. Zheng, “Ros navigation tuning guide,” in *Robot Operating System (ROS): The Complete Reference (Volume 6)*, A. Koubaa, Ed. Cham: Springer International Publishing, 2016, pp. 197–226.

Appendix A

# PDDL domain

```
(define (domain turtlebot3)

(:requirements :strips :typing :fluents :durative-actions)

(:types
  waypoint
  robot
)

(:predicates
  (robot_at ?v - robot ?wp - waypoint)
  (undocked ?v - robot)
  (docked ?v - robot)
  (charge_at ?wp - waypoint)
  (photographed ?wp - waypoint)
)

(:functions
  (distance ?wp1 ?wp2 - waypoint)
  (speed ?v - robot)
  (min_charge ?v - robot)
  (state_of_charge ?v - robot)
  (charging_rate ?v - robot)
  (discharge_rate ?v - robot)
  (docking_duration ?v - robot)
  (traveled ?v -robot)
)


; Move to any waypoint, avoiding terrain
(:durative-action goto_waypoint
  :parameters (?v - robot ?from ?to - waypoint)
  :duration (= ?duration (/ (distance ?from ?to)
                (speed ?v)))
  :condition (and
    (at start (robot_at ?v ?from))
    (at start (>= (- (state_of_charge ?v)(* (discharge_rate ?v) (distance ?from
    ?to))) (min_charge ?v)))
```

```
38      (over all (undocked ?v))
39      )
40    :effect (and
41      (at start (not (robot_at ?v ?from)))
42      (at end (decrease (state_of_charge ?v) (* (discharge_rate ?v) (distance ?
        from ?to))))
43      (at end (robot_at ?v ?to))
44      (at end (increase (traveled ?v) (distance ?from ?to)))
45      )
46  )
47
48  ; Docking to charger
49  (:durative-action dock
50    :parameters (?v - robot ?wp - waypoint)
51    :duration ( = ?duration (docking_duration ?v))
52    :condition (and
53      (at start (charge_at ?wp))
54      (over all (robot_at ?v ?wp))
55      (at start (undocked ?v))
56      )
57    :effect (and
58      (at end (docked ?v))
59      (at start (not (undocked ?v))))
60  )
61
62  ; Unocking from charger
63  (:durative-action undock
64    :parameters (?v - robot ?wp - waypoint)
65    :duration ( = ?duration (docking_duration ?v))
66    :condition (and
67      (at start (charge_at ?wp))
68      (over all (robot_at ?v ?wp))
69      (at start (docked ?v)))
70    :effect (and
71      (at start (not (docked ?v)))
72      (at end (undocked ?v)))
73  )
74
75  ; Charging battery
76  (:durative-action charge
77    :parameters (?v - robot ?wp - waypoint)
78    :duration ( = ?duration (* (charging_rate ?v) (- 100 (state_of_charge ?v))))
79    :condition (and
80      (at start (charge_at ?wp))
81      (at start (robot_at ?v ?wp))
82      (over all (docked ?v))
83      (at start (<= (state_of_charge ?v) 100)))
84    :effect (and
85      (at end (assign (state_of_charge ?v) 100))
86      )
87  )
88
89  ; Photographing an object of interest
90  (:durative-action inspect
91    :parameters (?v - robot ?wp - waypoint)
92    :duration ( = ?duration 10)
```

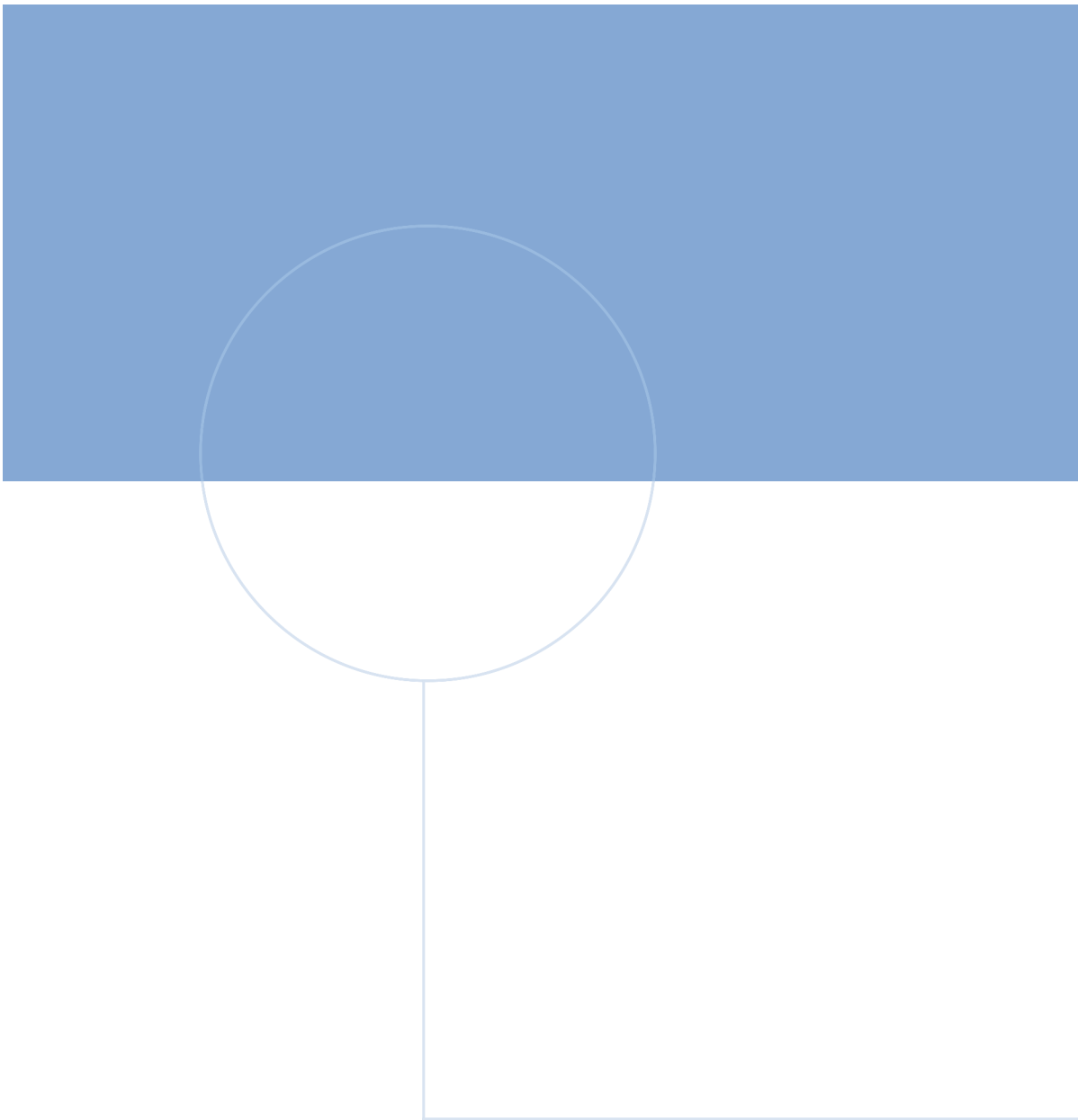
```
93    :condition (and
94      (over all (robot_at ?v ?wp))
95      (at start (>= (- (state_of_charge ?v) 3) (min_charge ?v))))
96    :effect (and
97      (at end (photographed ?wp))
98      (at end (decrease (state_of_charge ?v) 3))
99      )
100 )
101 )
```

Appendix B

# PDDL problem

```
(define (problem task)
(:domain turtlebot3)
(:objects
    wp0 wp1 wp2 wp3 wp4 wp5 wp6 wp7 - waypoint
    turtlebot - robot
)
(:init
    (robot_at turtlebot wp0)
    (undocked turtlebot)

    (charge_at wp0)
    (charge_at wp1)

    (= (distance wp1 wp0) 5.59464)
    (= (distance wp0 wp1) 5.59464)
    (= (distance wp2 wp0) 2.94109)
    (= (distance wp0 wp2) 2.94109)
    (= (distance wp2 wp1) 5.80086)
    (= (distance wp1 wp2) 5.80086)
    (= (distance wp3 wp0) 7.15122)
    (= (distance wp0 wp3) 7.15122)
    (= (distance wp3 wp1) 4.66476)
    (= (distance wp1 wp3) 4.66476)
    [...]         ; other distances removed to reduce length

    (= (speed turtlebot) 0.1)
    (= (min_charge turtlebot) 15)
    (= (state_of_charge turtlebot) 100)
    (= (charging_rate turtlebot) 0.5)
    (= (discharge_rate turtlebot) 3)
    (= (docking_duration turtlebot) 1)
    (= (traveled turtlebot) 0)

)
(:goal (and
    (photographed wp2)
    (photographed wp3)
```

```
    (photographed wp4)
    (photographed wp5)
    (photographed wp6)
    (photographed wp7)
))
(:metric minimize (traveled turtlebot))
)
```



NTNU

Norwegian University of
Science and Technology