RAM ROBOTICS AND MECHATRONICS

Towards increased autonomy of a rail-guided robot for the inspection of ballast water tanks

C. (Cees) Trouwborst

MSc Report

Committee:
Prof.dr.ir. S. Stramigioli
D.J. Borgerink, MSc
Dr.ir. T.J.A. de Vries
Dr. M. Poel

March 2017

005RAM2017
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA CTIT
BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

Summary

Ballast Water Tanks (BWTs) of large ships are required by law to be regularly inspected for corrosion, to ensure a sufficiently strong mechanical structure of the hull. Recognizing the health hazards for inspectors of BWTs, a rail-guided inspection platform was proposed by Christensen *et al.* in the Robots in Tanks project. As part of the SmartBot project, this platform was further developed into RoboShip platform.

Since reliable connectivity to the platform cannot be assumed in steel BWTs a certain level of autonomy is required, which is currently not present in the platform. Given that inspection tasks are not fully predictable, but dependent on knowledge gained on the heavily obstructed, partially known environment, introducing *full autonomy* is not realistic. Aiming for a blend of mainly *executive control* and aspects of *shared control with robot initiative* is most promising for the inspection field, since it allows for both a large range of robot activities and explicitly tapping into the human operator's experience.

To answer the question that results from these considerations,

> How to increase the autonomy of the RoboShip platform to the level of *(blended) executive control*, given the fundamental task- and environment uncertainty of the inspection domain?

key challenges in increasing the autonomy of the RoboShip inspection platform were defined in three categories: challenging aspects in the environment, challenges stemming from the inspection task and challenges stemming from the status of the RoboShip platform itself. All challenges were translated to specific requirements for the control architecture or the system as a whole.

Based on the defined requirements a design for the control architecture was proposed, in which a discrete planner is combined with a number of continuous control components. This thesis shows that discrete planning systems can be very effective in introducing autonomy to an inspection robot. By altering the existing TFD/M planner to allow for changes in the goal state requirements as a result of elementary sensing actions, new information on the environment can successfully be taken into account.

Additionally, it has been shown that by using a discrete planning system that supports *modules*, the semantic gap can be bridged and relevant details on the continuous domain can be incorporated into the highest, most abstract layer of planning and control.

While not a component of autonomy, significant efforts have been made in localization and arm control, to advance the state of these two important requirements.

The simulator created as part of this research proved to be a very useful tool in researching and validating abstract, high-level control logic.

We believe that the proposed architecture and accompanying tools form a solid foundation for robustly introducing autonomous features to the RoboShip platform.

Preface

This report and the work it describes conclude my master thesis project at the University of Twente. It was great working on the topic of intelligent systems for such a long time, and I would like to thank the Robotics and Mechatronics group for this opportunity.

In particular, I would like to thank Theo, Dian and Stefano, my official supervisors. I have enjoyed the numerous conversations on a large range of interesting subjects, including some related to robotics. I would also like to thank Jan Broenink and Mannes Poel, for their input and for playing an important role in grading my work.

I hope you enjoy reading this report. If anything needs further clarification, please reach out using the information below.

Kind regards,

Cees Trouwborst
ceestrouwborst@gmail.com
+31(0)641002540

Table of contents

1 Introduction

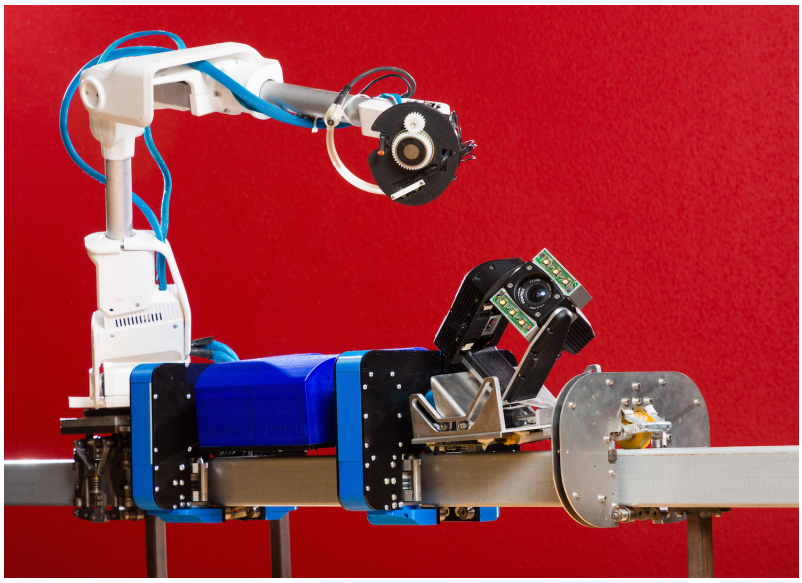


Figure 1.1: The current status of the RoboShip platform

Ballast Water Tanks (BWTs) of large ships are required by law to be regularly inspected for corrosion, to ensure a sufficiently strong mechanical structure of the hull. Recognizing the health hazards for inspectors of BWTs due to working in very humid, confined spaces, a rail-guided inspection platform was proposed by Christensen *et al.* in the Robots in Tanks project [Christensen et al. (2011)]. Apart from showing the economic feasibility of a robotic inspector, the paper shows a proof of concept rail-guided demonstrator, that has since evolved into the RoboShip platform shown in Figure 1.1, as part of the SmartBot project [SmartBot (2016)].

One of the requirements stated in the original paper is that the platform has to have a certain degree of autonomy, since reliable connectivity to the platform cannot be assumed in steel BWTs and thus constant teleoperation is not an option.

In addition, increasing the degree of autonomy beyond the required degree will increase the attractiveness of the platform for industry: if the cognitive load on the human inspector can be reduced, the human inspector can guide multiple robotic inspectors at once, reducing the total cost of maintenance.

These considerations directly result in the research question central to this thesis:

How can one transform the RoboShip platform to
an autonomous inspection system?

In order to answer this question, the term *autonomy* needs further specification. In the context of this thesis, the definition of autonomy by Franklin *et al.* is used:

> An *autonomous agent* is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future. —Franklin and Graesser (1997)

Figure 1.2: An example of a BWT with a large number of stiffener profiles

Loosely translated and projected on the inspection domain, this means that an autonomous system[1] is aware of its environment, actively takes action to learn about and interact with it, in order to pursue its purpose.

Different levels of robot autonomy can be considered using the taxonomy proposed by Beer *et al.* [Beer et al. (2014)]. In this taxonomy *full autonomy* is the highest level of autonomy, effectively fully replacing the human operator. This degree of autonomy seems unrealistic to aim for at this stage: introducing autonomy to the RoboShip platform is challenging because inspection tasks are not fully predictable, but dependent on knowledge gained on the heavily obstructed, partially known environment (Figure 1.2) *during* inspection.

We believe that a blend of mainly *executive control* and aspects of *shared control with robot initiative* is most promising for the inspection field. In *executive control* the robot only gets abstract, high-level goals and performs all subsequent activities autonomously. In *shared control with robot initiative* the robot is autonomous but can explicitly ask the human operator for help. This blended autonomy allows for both a large range of robot activities and explicitly tapping into the human operator's experience.

These definitions allow us to rephrase our research question more specifically:

> How can one increase the autonomy of the RoboShip platform to the level of *(blended) executive control*, given the fundamental task- and environment uncertainty of the inspection domain?

As a start, Chapter 2 provides relevant background on control architectures and planning in discrete and continuous domains. Chapter 3 will analyze the challenges of increasing the autonomy of the platform in more detail and list explicit requirements for the control architecture. Chapter 4 describes the proposed design, and Chapter 5 outlines an experiment validating this design. Chapter 6 summarizes the results of the experiments, which are evaluated in Chapter 7. Finally, conclusions and recommendations can be found in Chapter 8 and Chapter 9, respectively.

[1] The terms *autonomous agent, autonomous system* and *autonomous robot* will be used interchangeably in this thesis.

2 Background

This chapter covers background information that allows a well-founded discussion on autonomy in robotic systems. To give insight in the specific use case, Section 2.1 provides more information on the RoboShip platform and its subsystems. Section 2.2 details the history of commonly used control architectures. Section 2.3 provides the fundamentals of classical planning, and finally, Section 2.4 defines the field of *Integrated Task and Motion planning* and its core challenges.

This chapter is heavily influenced by a number of books, which are a great resource when researching autonomy in robotics [Ghallab et al. (2016); Murphy (2000); Thrun et al. (2005); LaValle (2006); Russell and Norvig (2009)].

2.1 Subsystems of the RoboShip platform

The RoboShip inspection platform consists of a number of subsystems, shown in Figure 2.1, that need to cooperate in order to perform an inspection. This section shortly elaborates on the subsystems and their role during operation.

The platform is rail mounted and locomotion is performed by two drive units (`E`). Camera (`F`) is mounted in Pan-Tilt-Unit `G` to perform a visual inspection. Bright LEDs are mounted on the camera to improve lighting conditions in the BWTs (`H`). To minimize the cross-sectional area perpendicular to the rail, which is of importance in obstructed environments, the camera can be stowed away using mechanism `I`.

When corroded spots are visually detected, additional inspection is required using inductive sensor `D`. Lightweight manipulator `B` is used to move close to the point of interest [Huttenhuis (2015)]. To counteract the flexibility that is present in both the manipulator arm and compliant rail, a specialized end-effector `C` was developed which effectively creates a two-stage robotic arm [Huttenhuis (2015); Borgerink et al. (2016); Ten Den (2016)]: the magnet in the end-effector allows for latching onto the environment, after which very accurate positioning of the sensor can be performed. Rotation around the axis of the rail during manipulator movement is minimalized using clamping mechanism `A`.

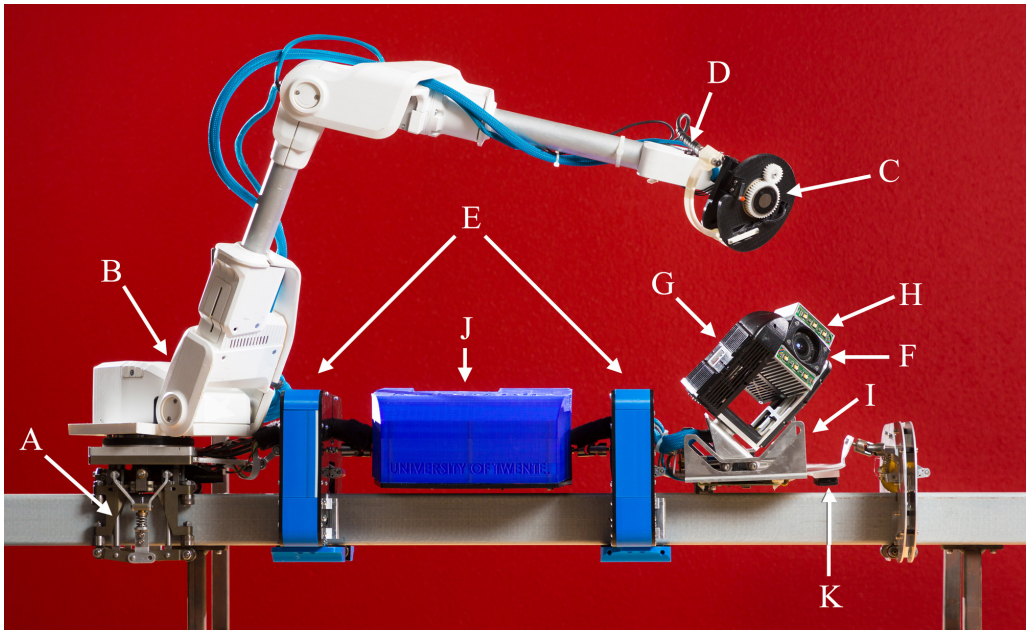
All subsystems connect to a computer located in `J`, using a number of methods and communication protocols (USB, Ethernet, RS485 and RS232) [Rijnbeek (2015)]. The computer connects to the outside world using WiFi. Also present in `J` are the batteries of the system, allowing for untethered operation, and an Inertial Measurement Sensor. A RFID tag reader is present and mounted at *K*. These sensor systems can aid in the localization of the robot in the tank.

In the current state, no autonomous features are present. All subsystems are controlled directly by a human operator, through general purpose controllers like gamepads and joysticks.

2.2 A brief history of control architectures

Shakey [Nilsson (1984)] (fig. 2.2) is considered to be the robotic system that boosted the research efforts into autonomous systems. With hardware that would be deemed extremely limited at this time, the robot was able to autonomously push objects from one room to another. When examining its control architecture, it can be classified as a “Sense-Plan-Act” architecture. The robot would take time to sense its environment, subsequently take time to plan, and finally act on this plan. While effective in static environments, such a control architecture is not able to function in dynamic environments [Firby (1987)].

The response of the field was to come up with more reactive control architectures. A direct connection between a sensor and actuator, through a relatively simple controller, would be



A	Clamping mechanism	G	Pan-Tilt-Unit
B	Manipulator arm	H	LEDs
C	End-effector with latching magnet	I	Stowing mechanism
D	Inductive coating thickness sensor	J	Computer
E	Drive unit	K	RFID tag reader
F	Camera		

Figure 2.1: The RoboShip robotic platform, with subsystems

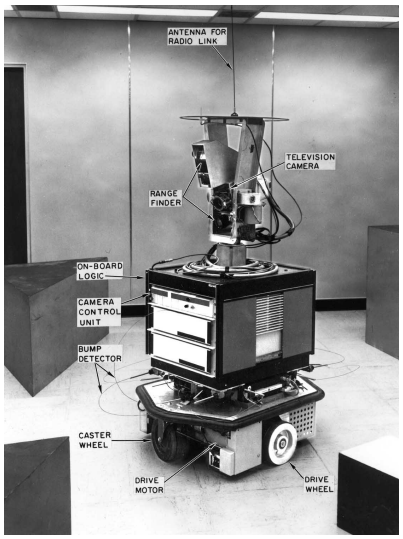
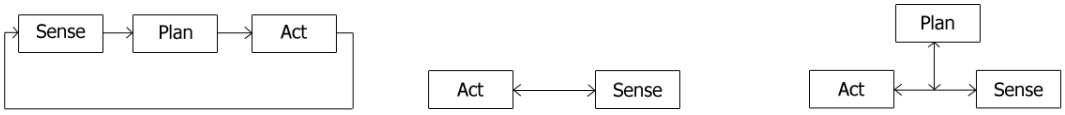


Figure 2.2: Shakey, the robot. Image courtesy of SRI International.



(a) Sense-Plan-Act approach **(b)** Reactive approach **(c)** Hybrid approach

Figure 2.3: Control architectures depicted schematically. The arrows can be interpreted as the possible interactions between core activities. For the hybrid approach, the planning component influences which behaviors are relevant given the current subtask.

called a "behavior". The output of different behaviors was combined to construct the final action of the robot [Firby (1987)]. This was partly inspired by nature: fundamental instincts of many insects seem to result in both reactive behavior (e.g. fleeing from a predator) and goal-directed behavior (e.g. a coordinated search for food). Ronald C. Arkin, one of the shapers of the field of behavior-based robotics, was particularly interested in the parallels between behaviors of animals and robots [Arkin (1998)].

While able to respond to changes in the environment adequately, implementing goal-directedness proved difficult [Gat (1997)]. Some tried to counteract this with smarter, hierarchic "merging" behavior, the so-called *subsumption* methods [Brooks (1986)], others designed an architecture that has an explicit split between deliberate, planned behavior and reactive behavior, so-called *hybrid* methods (e.g. Connell (1992)).

The *three layer* architecture can be considered a class of hybrid architectures that are characterized by having a reactive layer, a deliberate layer and a *sequencing* layer that connects the two [Gat (1997)] and allows for guiding reactive control. The three layer paradigm has been very popular ever since its conception.

Explicitly defining the roles of the layers helps in translating the problem to implementation requirements. It can be required that parts of the reactive layer function in real-time, while the deliberate layer does not have that requirement [Broenink and Ni (2012)].

The deliberate and sequencing layers can be matched to the problem at hand, to create an applicable multi-controller solution matching the complexity of the problem [Van Breemen and De Vries (2001)].

The different control architectures can be schematically depicted as seen in fig. 2.3 (adapted from Murphy (2000)).

2.3 Classical planning

This section formalizes the *deliberation* that (mainly) takes place in the deliberate layer of the control architecture. Since "classical" planning will be used in this thesis, this chapter will first elaborate on classical planning and its assumptions. Additional remarks on planning formalisms for when classical planning assumptions do not hold are elaborated on in the final paragraphs. The recent work of Ghallab *et al.* [Ghallab et al. (2016)] is used as a basis for this section and its notational conventions.

Definition 2.3.1. A *state-transition system* or *classical planning domain* is a 4-tuple $\Sigma = (S, A, \gamma, \text{cost})$, where

- S is a finite set of *states* in which the system may be.
- A is a finite set of *actions* that the system may perform.
- $\gamma : S \times A \rightarrow S$ is a partial function called the *prediction function* or *state transition function*. If $\gamma(s, a)$ is defined, a is *applicable* in s, with $\gamma(s, a)$ being the *predicted* outcome. Otherwise a is *inapplicable* in s.

- cost : $S \times A \rightarrow [0, \infty)$ is a partial function having the same domain as γ representing the costs that are to be minimized in the planning domain.

To be able to use Definition 2.3.1 as the deliberation component of an autonomous system, the following so-called *classical planning assumptions* need to be taken into consideration:

- *Finite, static environment* In addition to requiring the sets of states and actions to be finite, Definition 2.3.1 assumes that changes only occur in response to actions.
- *No time* There is no explicit model of time in this planning formalism.
- *Determinism* Definition 2.3.1 assumes that we can always predict with certainty what state will be produced if an action a is performed in a state s: there is no room for execution error and nondeterministic actions.

To further define the state, the following definitions are relevant:

Definition 2.3.2. A *state variable* is a syntactic term

$$x = sv(b_1, \ldots, b_k) \tag{2.1}$$

where sv is a symbol called the state variable's *name*, and each b_i is a member of set B, which is the set of names for all objects, plus any mathematical constants that may be needed to represent properties of those objects.

Definition 2.3.3. A *state-variable state space* is a set S of variable-assignment functions over some set of state variables X. Each variable-assignment function in S is called a *state* in S.

Definitions 2.3.2 and 2.3.3 show that the size of set S is dependent on the number of defined state variables, which in turn depends on the number of objects (e.g. robots, compartments, rusty spots, inspection locations) and corresponding properties defined. Besides B, set R represents planning domain Σ's rigid properties (never changing relations)

In order to formulate *actions*, the following definition on *atoms* is required:

Definition 2.3.4. A *positive literal* or *atom* (short for *atomic formula*), is an expression having either of the following forms:

$$rel(z_1, \ldots, z_n) \quad \text{or} \quad sv(z_1, \ldots, z_n) = z_0 \tag{2.2}$$

where rel is the name of rigid property in set R, sv is a state-variable name, and each z_i is either a mathematical variable or the name of an object. A *negative literal* is an expression having either of the following forms:

$$\neg rel(z_1, \ldots, z_n) \quad \text{or} \quad sv(z_1, \ldots, z_n) \neq z_0 \tag{2.3}$$

This allows us to define an *action template* as follows:

Definition 2.3.5. An *action template* for S is a tuple $\alpha = (\text{head}(\alpha), \text{pre}(\alpha), \text{eff}(\alpha), \text{cost}(\alpha))$, where

- head(α) is a syntactic expression of the form $act(z_1, \ldots, z_n)$, where act is the *action name* and $z_1, \ldots, z_n$ are the parameters of the action template.
- pre(α) = $p_1, \ldots, p_m$ is a set of preconditions, each of which is a literal (atom).
- eff(α) = $e_1, \ldots, e_o$ is a set of effects, each of the form $sv(t_1, \ldots, t_p) \leftarrow t_0$
- cost(α) is a number $c > 0$ denoting the cost of applying the action. This can be a function of the parameters.

If the combination of state s and rigid property literals in R satisfies pre(α), action a is an instance of α and *applicable* in s, and the predicted outcome is the current state, plus the effects described by eff(a).

Using these definitions, the planning problem P can be expressed as looking for a sequence of actions $\langle a_1, \ldots, a_n \rangle$ that results in an end state satisfying the set of goal literals g, given initial state s_0 in planning domain Σ. *Planning* can be defined as the search for this sequence, or *plan* using search algorithms. Going into search algorithms is beyond the scope of this thesis, but the following is good to know:

- *Forward* search algorithms start with the initial state and systematically try different actions that are applicable. A popular example is Fast-Forward [Hoffmann and Nebel (2001)].
- *Backward* search algorithms start with the goal state and systematically try different actions that result in that state or the relevant intermediate state. An example is pre-image backchaining [Kaelbling and Lozano-Pérez (2013)].
- *Hybrid* search algorithms try to combine a forward and backward search algorithm. An example is hybrid backward-forward search [Garrett et al. (2015)].
- *Heuristics* can be used to guide the search, preventing the need to explore the whole state space. Heuristics can be domain-dependent, domain-independent and/or based on the cost function. An example is the *greedy* heuristic, which makes the best local decision in each search step, in the hope to find a global solution.
- Using backward search algorithms makes it impossible (or very difficult) to use a cost function that is dependent on the state.

If actions have a known probability of success instead of deterministic outcome, or the state is partially observable, the formalism above does not allow for an accurate representation of the problem at hand. Markov Decision Processes (MDPs) allow for modeling the uncertainty in the outcome of actions. Partially Observable Markov Decision Process (POMDPs) explicitly model the role of observations, as well. [Ghallab et al. (2016)]

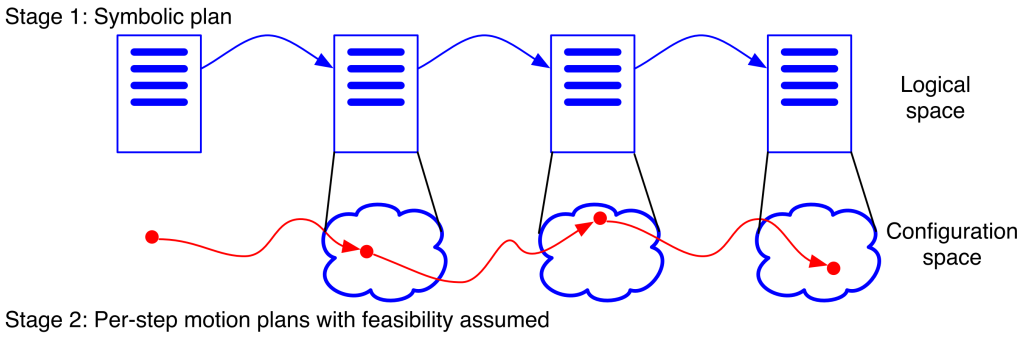
A different approach to handling uncertainty or unpredictable events is to carefully monitor the plan, and check if the current plan still results in the goal state. If not, a new plan should be made. While less elegant and non-optimal, this approach decreases the modeling efforts significantly.

2.4 Integrated task and motion planning

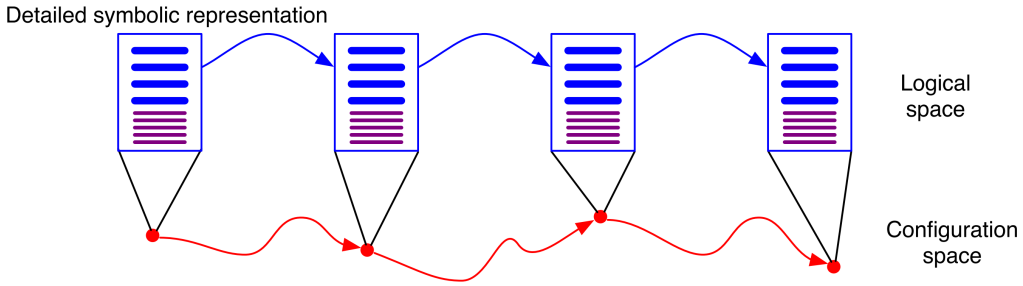
While the previous section describes how decision making can work on the highest level of control, the information available to that control layer is usually not sufficient for lower level controllers, creating a *semantic gap*. This is a result of the fact that the sets used in planning are finite (and preferably as small as possible), while the real world is continuous and therefore contains an infinite number of possible properties for an object. The position of the RoboShip base can be considered an example: while only a finite number of positions on the rail is relevant for performing an inspection, the possible positions, and thus the search space, is infinite. Handling this challenge is at the core of the field of *integrated task and motion planning*.

A straight-forward approach to this is to discretize the workspace, limiting the search space for the discrete planner. This approach might still result in discrete domains with thousands of configuration symbols, however, making planning infeasible [Srivastava et al. (2013)]. It also ignores the fact that very efficient, domain-specific planning algorithms are available for planning in continuous space.

As as an alternative, an interface between discrete planning and continuous-space (or geometric) planning can be introduced (e.g. Srivastava et al. (2013); Alami et al. (1998); Kaelbling and Lozano-Pérez (2013); Dornhege et al. (2009)). While the discrete planner deals with the (discrete) space of logical representations, the geometric planners can deal with the (continuous) configuration space. A detailed description of both domains and an overview of



(a) Sense-Plan-Act (as used in [Nilsson (1984)]): Planning in the configuration space takes place completely independent from planning in the logical domain. The congruity of the two spaces is assumed.



(b) Semantic Attachements (as used in [Dornhege et al. (2009)]): Relevant information on the configuration space is stored in the logical space (indicated in purple) and can be used in validation and as parameters for actions.

Figure 2.4: Schematic representation of two approaches to connecting the discrete logical and continuous geometric domain. Image courtesy of Kaelbling and Lozano-Perez (2012)

.

several approaches to the implementation of such an interface is given in the work of Kaelbling *et al.* [Kaelbling and Lozano-Perez (2012)].

Figure 2.4 shows two schematic representations of the search through the two aforementioned domains, as an example. Figure 2.4a shows the approach taken in a Sense-Plan-Act control architecture. The search in the logical and configuration space are fully decoupled, and feasibility of the motion plans is assumed. Figure 2.4b shows an approach in which very specific geometric information is added to the logical domain, supplied by geometric planners and indicated in purple, for which it is known that feasible motion plans exist.

Interface calls between the discrete and geometric planners are often called *external modules*, or *external atoms*. The work of Erdem *et al.* [Erdem et al. (2016)] shows that the use of such interfaces is most effective when the modules for planning in the geometric space can be called *during* the search in the logical domain, which requires modification of the implementation of the planner (e.g. Dornhege (2015); Trüg (2006)).

3 Problem analysis

This chapter elaborates on the challenge of transforming the RoboShip platform to an autonomous system. Not all aspects of this transformation are directly related to autonomy; some can be interpreted as required conditions for implementing autonomy.

Three main categories are relevant: challenges related to the environment, the task and the platform itself. Apart from those predictable challenges, a category that can be described as "Unknown uncertainty" is relevant as well. Table 3.1 summarizes the different aspects of the problem, that are described in more detail and translated to requirements in the following sections.

3.1 Environment and environment sensing

The environment where the robot resides in poses a number of challenges. Those challenges are either directly related to the environment, or to the robots capability of sensing the environment.

Environment geometry

The geometry of the environment can be considered static, since detailed CAD files are usually available for the ballast water tanks. Since the manipulator is specifically designed for this environment [Huttenhuis (2015)], it is assumed all relevant locations in the tank can be reached. Taking into account the complexity of the tank environment (see Figure 3.1), the platform has to be able to control the arm in a way that avoids collisions with the rail, features in the environment and itself.

It should be mentioned that there is always the possibility that the tank geometry is different from the definition. The proposed control architecture must be able to handle this kind of uncertainty.

The control architecture should be able to perform collision-free manipulation planning in a known environment featuring a large number of obstacles.

Localization

No robust localization algorithm has been implemented and validated yet on the RoboShip platform, despite multiple projects on the topic [Abdelhady (2016), Gies (2015)]. A robust localization solution is required for introducing autonomy, since the pose estimate directly influences the estimate of the distance between the robot and its environment. In the following paragraphs, challenges in interpreting data from the available sensors are shortly summarized.

While the environment is static, the robot moves through it over a rail that can be slippery, effectively reducing traction and possibly rendering the information from wheel-encoders

Table 3.1: Different aspects of the problem

Environment-related	Task-related	Platform-related
Geometry	Dynamic goal state	Robust software
Localization	Visual rust detection	State estimation
Unreliable connectivity	Manipulator movement	Battery-powered
	Unknown uncertainty	

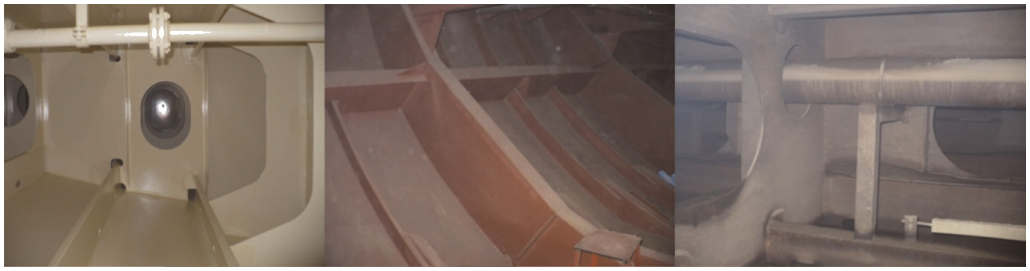


Figure 3.1: A number of ballast water tanks, indicating the complexity of the environment. Image courtesy of Christensen et al. (2011)

.

invalid. Dirt attached to the rails might even make movement impossible. RFID tags on the rails can be used as landmarks to correct for errors using filtering [Thrun et al. (2005)].

Besides RFID tags, features in the environment can act as landmarks for visual localization algorithms. The highly repetitive nature of some of these features, e.g. the stiffener profiles, might introduce difficulties, however. Since the environment is mostly made of steel, orientation estimates from an Inertial Measurement Unit will be inaccurate. This feature can be used to our benefit as well when using the distorted earth magnetic field as a map [Christensen et al. (2011)].

Another aspect that influences the accuracy of the localization algorithm, is that the rail has a significant compliance [Borgerink et al. (2014)]. This means a position on the rail cannot be directly translated to a position in 3D and has led to the decision to use a two-stage manipulator [Borgerink et al. (2016)].

With respect to autonomy, we require that *the location of the robot in the tank should be estimated with an accuracy of approximately 1 cm.*[1]

Connectivity

The lack of reliable communication between the operator and the robotic platform was the main reason for requiring a certain degree of autonomy [Christensen et al. (2011)]. This can be directly related to the large amount of metal in the environment, which makes WiFi communication less reliable.

The control architecture must be able to plan for an inspection and its execution, even when loss of signal occurs regularly.

3.2 Task definition

The inspection task itself is challenging and not completely predefinable. To be able to pinpoint challenges, this section will start by defining an inspection task.

The *inspection task* is defined as follows:

> When performing an inspection task, the robot is placed on the rail of a known ballast water tank. The rail cannot branch off, so the robot does not have the freedom to choose a path through the tank. The robot has no knowledge about the state of the tank and the presence of corrosion.
>
> The robot will plan to visually inspect all compartments of the tank, and start executing the visual inspection. During the visual inspection, corroded sections can be detected that need further inspection. The robot will plan for the

[1] None of the projects on localization formally define a requirement for the localization component. This requirement is an estimate based on performance requirements from Ten Den (2016) and Overbeek (2015)

required manipulator movement[2] and take measurements where necessary using the inductive sensor. All data is saved for interpretation by the human operator after the inspection.

Dynamic goal state

When relating the inspection task to classical planning methods described in Section 2.3, it can be observed that the goal state is not static in the inspection domain. The effect of the visual inspection can be described in the goal state at the start of the inspection task (e.g. "all compartments seen"), but the inspection with the sensor cannot, since the planner does not know about any points of interest that need inspection. Observations from looking around result in new objects in the symbolic state, and newly introduced predicates in the goal state.

While this is a good match with the definition of an autonomous agent (*A system (..) that senses that environment (..) over time, in pursuit of its own agenda.* – Franklin and Graesser (1997)), this observation results in additional planning and plan monitoring.

The planning component must be able to recognize that the current plan is not effective given new information on the goal state and trigger replanning.

Rust detection

Since rust detection is at the very core of the task, it should be performed in a robust manner. However: visual recognition of rusty spots in a ballast water tank is not trivial. Visibility is usually low because of bad lighting and the large number of metal obstacles.

While it is expected that it is possible to train classifiers using machine learning algorithms, said classifiers are not available yet. The autonomous platform should be able to handle this uncertainty.

The robotic platform should be able to accurately classify rust and report on the accuracy of its classification.

Manipulator movement

In addition to the requirements that the environment imposes on the movement of the manipulator, the inductive coating thickness sensor requires positioning perpendicular to the surface of interest with an accuracy within 5° [Huttenhuis (2015)], effectively requiring a geometric planner that can handle a 5-DOF setpoint in Cartesian space.

The geometric planner should be able to handle partial setpoints in Cartesian space.

3.3 Challenging aspects of the RoboShip platform

Challenges in introducing autonomy can stem from the design or current status of the platform as well.

Robust software

Introducing autonomy requires the composition of small, specialized software packages that have a clearly defined scope and functionality. In its initial state, the available software lacked the required documentation of interfaces and was largely untested. In addition, some required functionality was not yet implemented (e.g. the automated detection of rust or manipulator control with obstacle avoidance).

The software interfacing with the different subsystems should be robust, documented and tested.

[2] Since the manipulator was still under active development using this thesis, performing an inspection is simplified to moving the arm to a relevant end-effector pose.

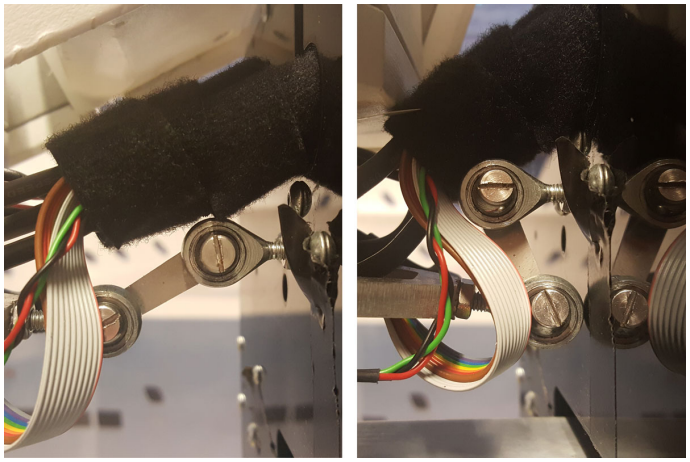


Figure 3.2: Joint with multiple, unobservable states

State estimation

The RoboShip platform consists of a number of carts fixed together using universal joints. The exact orientation of the carts with respect to each other is unknown, since no sensors are present that provide that information.

The joint connecting the manipulator to the drive unit is not a universal joint, but a combination of two ball joints that has two (unmeasurable) "equilibrium states", that can both exist during base movement (see Figure 3.2). This results in two options for the transformation between the drive unit and base of the manipulator.

The inaccuracies in the state estimation described above propagate to the state estimation of the manipulator w.r.t the environment, which can, in turn, result in unexpected collisions.

The full state of the system should be observable.

Battery-powered

The platform is battery powered and therefore effectively range- and time limited. As an autonomous system, the platform should monitor itself and take the remaining battery charge into account during planning and execution.

The control architecture must take into account that the platform is battery powered and thus limited in range and operation time.

3.4 Unknown uncertainty

The capability of human beings to handle situations that they have never seen before or learned about is unparalleled by autonomous systems. This poses an additional challenge when designing autonomous systems: how to design for situations that the designer does not know about? While this challenge might require discussions of a more philosophical nature, *the proposed design must at least result in predictable behavior when detecting non-nominal conditions.*

4 Design

The previous chapter detailed the major challenges in transforming the RoboShip platform into an autonomous agent. This chapter proposes a control architecture that allows for handling those challenges.

Since the overall control architecture is at the core of this thesis, not all of the subsystems will be implemented. Table 4.1 gives an overview of the aspects of the problem, annotated for their role in this thesis.

Since autonomous features can be considered high-level control, a discrete planning algorithm is at the core of the proposed design, shown schematically in Figure 4.1. Given that the domain is relatively static and costs are important in choosing a solution, we opt for a solution with a forward-search classical planner. In accordance with Section 2.3, we recognize that a reliable monitoring mechanism is needed to manage a dynamically changing goal state requirement.

The discrete planner should be able to intertwine geometric planning and symbolic planning, given that manipulator movement in a heavily obstructed environment is required. Following the recommendations of Erdem *et al.* [Erdem et al. (2016)], a planner is chosen that supports external atoms that are evaluated during planning. Specifically, the work of Dornhege [Dornhege (2015)] was extended to support a changing goal state. It features the Temporal Fast Downward with Modules planner (TFD/M), which is a modified version of the Temporal Fast Downward planner [Eyerich et al. (2012)] that allows interfacing to external libraries using *modules*. The language used to describe the domain is PDDL/M [Dornhege et al. (2009)].

The details of the proposed design are described in the remainder of this chapter. Section 4.1 describes the hybrid symbolic state used in planning. Section 4.2 describes the elementary actions that are to be described in the PDDL/M description language. The full PDDL/M domain description file, containing both the actions and the possible components of the state, can be found in Appendix D. While the contents of these two sections focus on inspection tasks, Section 4.3 proposes to extend the symbolic state to manage uncertainty.

4.1 Symbolic state description

The symbolic state changes over time based on sensor measurements, which are the results of elementary actions. When given a goal state requirement, the planner makes a plan to get from the current state to a goal state.

Table 4.1: Different aspects of the problem, annotated for the approach taken in this thesis

Environment-related		**Task-related**		**Platform-related**	
Geometry	++	Dynamic goal state	++	Robust software	++
Localization	+	Visual rust detection	++, o	State estimation	o
Unreliable connectivity	+	Manipulator movement	++	Battery-powered	+
		Unknown uncertainty	+		

++ This thesis proposes and (partly) implements a design to counteract this problem.
+ This thesis proposes a design to counteract this problem.
o This thesis assumes a working implementation.

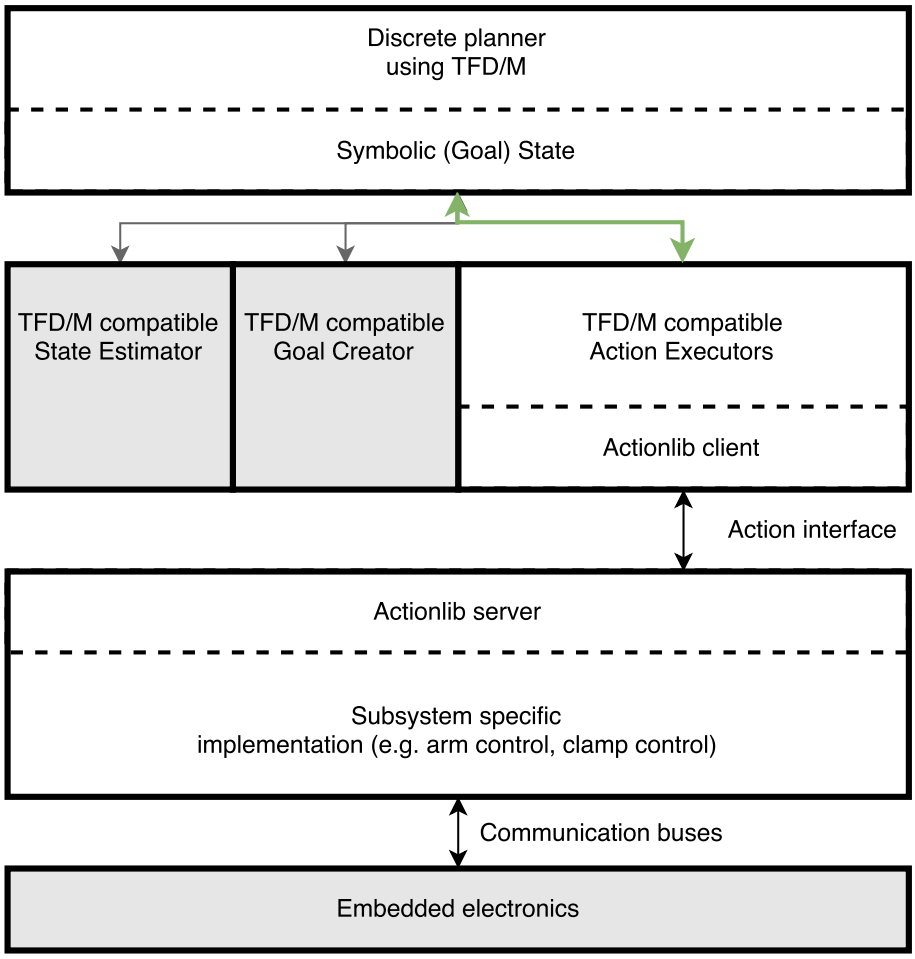


Figure 4.1: Proposed control architecture. Parts shown in gray are not considered in detail in this thesis. The relation shown in green is extended in comparison to the original implementation.

Table 4.2: Elements of the symbolic state

Type	**Supertype**	**Attributes** (PDDL/M functions)	**Role**
robot			Represents the robot
dist		*d* (m) distance	Point on the rail, described by distance travelled from the start
idist	dist		Point on the rail, used as starting point for an inspection
pose		**x** (m) 3D point in space, **q** quaternion representation of orientation	Pose in 3D space
rust	pose	p (%) estimated probability of rust present	Location of rust

Table 4.3: Boolean predicates of the symbolic state

Predicate	**Arguments**	**Role**
folded	robot *r*	Describes if the manipulator of robot *r* is folded.
stowed	robot *r*	Describes if the PTU of robot *r* is stowed away or in panoramic mode.
clamped	robot *r*	Describes if the clamping mechanism of robot *r* is activated.
lights	robot *r*	Describes if the lights on the PTU of robot *r* are activated.
seen	dist *d*	Describes if position on rail *d* was used for looking around.
inspected	rust *r*	Describes if rusty spot *r* was inspected using the manipulator.
at-location	robot *r*, dist *d*	Describes if robot *r* is at location *d*.

Table 4.2 shows the objects that can exist in the symbolic state. These objects can have multiple numeric attributes, allowing them to carry parameters necessary for geometric reasoning that occurs in actions and modules.

Table 4.3 shows the boolean predicates that exist in the symbolic state. When not explicitly set during state estimation, boolean predicates are considered unknown. To allow planning, the effects of elementary actions are expressed as the effects on the boolean predicates.

The goal state can be described by these predicates as follows:

1. All *dist* objects are *seen*, assuming the state only consists *dist*s relevant for looking around.
2. All existing *rust* objects are *inspected*.
3. The robot is *stowed* and *folded*.
4. The robot does not have activated *lights* and is not *clamped*.
5. *(Optional)* The robot is at a pre-defined final position.

4.2 Elementary actions

Table 4.4 summarizes the elementary actions that the RoboShip platform needs to be capable of in order to perform an inspection. Arguments or parameters of these actions are not listed for readability (they can be found in Appendix D), but are of course of importance. As an example, key actions *move arm* and *look around* are elaborated upon in the following sections.

In general, actions have a duration and the planner looks for the solution that has the lowest total time. The duration of an action can be calculated using a *module* to reflect e.g. the time it takes to move from one point to another. In this thesis, such a module is implemented for the movement of the robotic base over the rail.

4.2.1 Move arm action

Parameters: robot *r*, rust *p*, idist *d*

The *move arm* action moves the end effector of the manipulator to a given pose. Doing this requires calculating a trajectory from the current state to a goal state that has the end-effector in the desired configuration, which is one of the solutions to the manipulator's inverse kinematics problem. Checking if such a solution exists is done by a module that supplies a literal to the precondition of the action.

In order to be able to prevent collisions, the location of the robot on the rail is relevant and thus used as input through the position parameter of object *d*, which has type *idist*. For each point of interest, an infinite number of *idist* objects will be effective since *idist* objects have numerical attributes that describe the continuous domain. To decrease the size of the search space significantly, only relevant *idist* objects are added to the symbolic state by the *look around* action, describing the locations on the rail that are closest to points of interest. [1]

Since the planner will call the module often during planning and inverse kinematics calculations are costly, it is worthwile to look into optimizations and shortcuts in the precondition module. While less generic, these simplifications might result in significant performance gains. A trivial simplification is that the inverse kinematics solver does not need to be called if the point is outside of the workspace of the manipulator.

4.2.2 Look around

Parameters: robot *r*, dist *d*

The *look around* action moves the Pan-Tilt-Unit such that the camera will capture images of the whole tank. These camera images are then automatically scanned for rust. If rust is detected, the exact position of the rust is determined using *raycasting* based on the current pose of the camera and knowledge on the environment. For each of the rusty spots detected, both the location of the rusty spot and a candidate rail location suitable for inspection are added to the symbolic state.

The challenge lies in determining at which distances on the rail (described by *dist* objects) the robot should look around. Since looking around is relatively cheap and fast, a simple discretization is used: in every compartment of the tank, two locations are chosen for *look around*. In a more complex environment, a module can be implemented that uses coarse raycasting to validate if the view cones in the trajectory cover all relevant surfaces (as in Kaelbling and Lozano-Pérez (2013)).

4.3 Handling uncertainty

Uncertainty is present, since we are building a robot that interacts with a real-world environment. Examples in the RoboShip scope are uncertainty that an observed pattern is actually rust, uncertainty that the end effector can successfully reach a tricky spot in the tank, uncertainty that the tank specification is correct, and uncertainty that the current location is correctly estimated by the localization algorithm. Since it is impossible to take these uncertainties away, we must define a way to handle them.

Given that an inspection robot is usually accompanied by a human operator (even when autonomous features do the heavy lifting), a robot-human team is envisioned, where the human operator can act as an expert and guide the robot in nontrivial situations.

[1] It is good to observe that this assumption can result in unsolvable problem definitions, if no inverse kinematic solutions exist for the closest location on the rails for a point of interest. This is not the case in our domain. If this would be the case, a smarter *suggestor* module should be implemented which interfaces with the inverse kinematics calculations directly.

Table 4.4: Elementary actions for the RoboShip platform

Name	Effect	Conditions
Move base	Moves the base of the robot	Folded arm, unactivated clamp
Move arm	Moves the arm of the robot to a given pose	Activated clamp, goal pose reachable
Look around	Uses the Pan-Tilt-Unit to look around for corroded spots	PTU in panoramic mode, lights on
Fold arm	Fold the arm in	
(Un)clamp	(De)activate the clamping mechanism	
Lights on/off	Turns the lights on the camera on or off	
(Un)stow	Puts the PTU in stowed or unstowed (panoramic) mode	

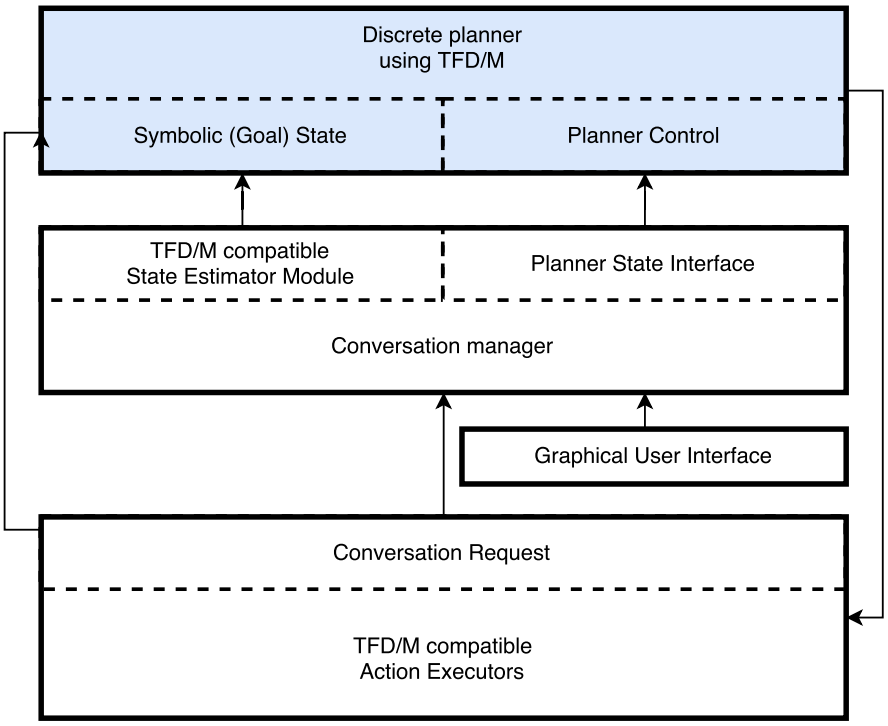


Figure 4.2: Proposed approach for including a human operator in the deliberation process. Parts shown in blue have seen significant development in this thesis.

A conceptual sketch of a possible implementation can be seen in Figure 4.2. As an illustrative example, the detection of a pattern that *possibly* may be rust is elaborated upon.

- The *look around* action recognizes the point of interest, and decides that the spot needs confirmation based on a pre-defined certainty band. It inserts the *rust* spot into the symbolic state, but does not add the *inspected* literal for the object to the goal state.
- The *look around* action also requests a new conversation from the *conversation manager*. It includes information relevant specific for the action, like the estimated probability and a picture of the point of interest. It also includes information on the impact of the conversation: can operation continue, or should the planner and execution be halted? Conversations can include a deadline, after which execution will be halted.
- The *conversation manager* uses its interface to the planner to pause execution if necessary. In this case, execution is temporarily paused (to allow adequate handling of a quick response from the operator) and then continued.
- The human operator can chime in on the conversation through the Graphical User Interface at a convenient moment when communication is available. Depending on the level of abstraction implemented for the specific type of conversation, the operator can change literals directly or give task level feedback. In this case, the operator chooses that the spot needs inspection.
- The conversation manager translates the response of the operator to the required change in the goal state, and applies that change through the *state estimator module*: the literal *inspected* needs to be added for the object added in the first step.
- Since the current plan does not result in the goal state, replanning is triggered by the monitoring process of TFD/M.
- The conversation is closed.

The conversation manager can log queries and responses for analysis, resulting in a structured dataset that can be used for supervised learning.

On the implementation, two remarks should be made:

- Manipulating the goal state can result in unsolvable plans. The Planner State Interface should be able to recognize this and pause execution.
- Input from the human operator might change the goal of the inspection as a whole. As an example: if the tank is too hazardous for the robot, the only objective might become to successfully leave the tank. This means that the GUI should be able to change the goal state requirements.
- It depends on the urgency of the question if the robot will continue with its plan, temporarily stop or fully abort. If a question is critical but a connection to the human operator is not available, logic can be added to move to the last location where communication was possible.

Given the time constraints for this thesis, the Conversation Manager and related interfaces were not implemented. This section does, however, show that, conceptually, a human-robot team can be created in the proposed control architecture in order to handle uncertainty.

4.4 Implementation

Each of the elementary actions is implemented as a ROS *actionlib*[2] client that can be used by the planner. The *actionlib* client connects to a corresponding *actionlib* server, which can be implemented for the real robotic platform as well as for simulation software, as long as the interface is equivalent. The main components of these interfaces are the custom ROS message types they consist of. As an example, the interface for the *move arm* action is described in Table 4.5.

[2] http://wiki.ros.org/actionlib

Table 4.5: Relevant components of the *move arm* action interface

Goal description	
`frame` (string)	reference frame used for coordinate
x (3D vector (m))	position w.r.t. reference frame
q (quaternion)	orientation w.r.t. reference frame
Result description	
`success` (boolean)	`true` if action successfully completed
Status description	
`active` (boolean)	`true` if action is running

When a similar design is used for the sensors, the autonomy of the system is implemented in a robot agnostic way[3].

All required *actionlib* servers are defined in a large number of ROS packages, since the action servers are implemented for a specific piece of hardware or functionality. E.g.: the `rs_base` package should provide an action server for the *move base* elementary action. The software package implementing the simulation is considered one piece of functionality, and therefore contains a large number of action server implementations.

An overview of the efforts related to software development performed as part of this thesis can be found in Appendix E.

4.5 Evaluation of the requirements

In order to describe specific design choices, the requirements stemming from the different aspects of the problem are revisited. Each of the following subsections will summarize the respective requirement and the implications on the design.

4.5.1 Environment Geometry

Requirement

The control architecture should be able to perform manipulation planning in a known environment featuring a large number of obstacles.

Scope

This thesis proposes and (partly) implements a design to counteract this problem.

Approach

While arm control of the manipulator was the focus of an earlier project [Overbeek (2015)], this work was not yet integrated into the software framework used in the RoboShip project. Additionally, the existing control solution does not include any collision avoidance.

As an alternative, ROS MoveIt! was implemented[4]. MoveIt! streamlines the process of modeling and controlling a robotic manipulator, includes tools for visualizations and provides interfaces to a number of geometric planning algorithms. Obstacle avoidance is provided out-of-the-box, and the software proved effective in other projects at the Robotics and Mechatronics group (e.g. Barbieri (2016)). For more information on the implementation of control for the manipulator, see Appendix C.

[3] Naturally, one must take care that the simulated system and environment match the real world system and environment: e.g. robot dimensions, actuator setpoint units, and gearing must be equal.

[4] http://moveit.ros.org/

The use of modules allows us to interface our discrete planner to the continuous planning capabilities of MoveIt!. Collision objects do not need to be stored in the symbolic state, but are stored in MoveIt!'s planning scene using an OctoMap representation [Hornung et al. (2013)].

At this point, no explicit design is proposed for detecting and handling incorrect knowledge on the environment or deflection of the rail. These factors should however only influence the state estimation based on sensor data, which is possible in the proposed architecture.

4.5.2 Localization

Requirement

The location of the robot in the tank should be estimated with an accuracy of approximately 1 cm.

Scope

This thesis proposes a design to counteract this problem.

Approach

Accurate localization of the robot on the rail is a key requirement for autonomy. It is, however, considered out-of-scope w.r.t. this thesis. To understand the challenge to the fullest, some work was done on localization, based on which a design was proposed. These efforts are described in Chapter A, and can be summarized as implementing a particle filter based on wheel encoder data and known RFID- and orientation landmarks. The design was partially implemented and needs further testing and calibration.

Multiple approaches can be taken for integrating this work into the larger control architecture. Since the base is connected very tightly to the rail, it is assumed that variations in pose are negligible when the robot is not moving deliberately. If large variations are present, the monitoring process will make sure the plan is invalidated and replanning is triggered.

4.5.3 Unreliable connectivity

Requirement

The control architecture must be able to plan for an inspection and its execution, even when loss of signal occurs regularly.

Scope

This thesis proposes a design to counteract this problem.

Approach

The proposed control architecture assumes that the system is entirely autonomous, but can pose questions to an expert if necessary. As is described in Section 4.3, it depends on the nature of the question if the robot will continue, pause, halt or abort. In addition, the proposed architecture allows for requiring connectivity for certain actions.

4.5.4 Dynamic goal state

Requirement

The planning component must be able to recognize that the current plan is not effective given new information on the goal state, and trigger replanning.

Scope

This thesis proposes and (partly) implements a design to counteract this problem.

Figure 4.3: Challenging geometry for rust inspection. Image courtesy of Warre te Riet o.g. Scholten.

Approach

The TFD/M planner implementation used as a basis for the deliberative layer [Dornhege (2015)] did not allow for dynamically changing the goal state. It was modified to allow interaction with the goal state as a result of actions, as is the case for the *look around* action. The monitoring step of the execution algorithm triggers replanning if the current plan is not effective.

4.5.5 Visual rust detection

Requirement

The robotic platform should be able to accurately classify rust and report on the accuracy of its classification.

Scope

This thesis proposes a design to counteract this problem. Everything related to visual inspection of actual rust is considered out-of-scope.

Approach

Visual detection of rust is considered out-of-scope. As a replacement, QR codes are detected. To emulate a classification algorithm, we embed information on the probability of a correct classification in the QR code. The probability estimate can be used to decide if questions need to be posed to a human expert (see Section 4.3). This explicit interface to an expert can automatically create a dataset for training algorithms using supervised learning.

Since the geometry of the environment can make it impossible to perform an inspection using the manipulator (Figure 4.3), visual inspection is the only possibility in some situations. Therefore, all data should be saved for later reference.

4.5.6 Manipulator movement

Requirement

The geometric planner should be able to handle partial setpoints in Cartesian space.

Scope

This thesis proposes and (partly) implements a design to counteract this problem.

Approach

When using end effector setpoints in Cartesian space, MoveIt! requires specifications of all six degrees of freedom. While it is technically possible to allow large errors on one of the rotational degrees of freedom, this approach does not elegantly reflect the planning problem. In the current implementation, a fixed orientation is assumed. The mechanical design of the arm allows for this, without a decreased workspace.

ROS package `descartes`[5] aims to implement planning in Cartesian space and allows for partially defined setpoints. The MoveIt! Maintainers intend to integrate the `descartes` package into MoveIt!, making MoveIt! an attractive choice as the framework for manipulation planning.[6]

4.5.7 Robust software

Requirement

The software interfacing with the different subsystems should be robust, documented and tested.

Scope

This thesis proposes and (partly) implements a design to counteract this problem.

Approach

While software development should not be at the center of a research thesis, development should adhere to some best practices regarding software development. In summary, the following points of attention were closely monitored:

- All software should be under source control.
- All dependencies should be explicit and documented. This is tested automatically using Continuous Integration methods.
- All interfaces (and preferably all internal APIs as well) should be documented.
- Software should be divided in reusable, functional blocks.
- Testing should be possible without deployment to real hardware, to reduce the risk of breaking hardware and shorten the development cycle. For this, a simulation environment was developed (see Appendix B).

An overview of the efforts related to software development performed as part of this thesis can be found in Appendix E.

4.5.8 State estimation

Requirement

The full state of the system should be observable.

Scope

This thesis assumes a working implementation.

Approach

The fact that some of the joints are not observable, but do influence state estimation through error propagation, is a problem that should be solved in the hardware in the next iteration of development.

[5] http://wiki.ros.org/descartes
[6] https://discourse.ros.org/t/maintainer-meeting-recap-march-7th/1442/1

4.5.9 Battery powered

Requirement

The control architecture must take into account that the platform is battery powered and thus limited in range and operation time.

Scope

This thesis proposes a design to counteract this problem.

Approach

Since the planner that is at the core of the deliberate layer is a *temporal* planner, explicit modeling of time is performed during planning. Given that the elementary actions are correctly modeled using durative actions in PDDL/M, an estimation of the total required time is available. If this time is longer than the expected lasting battery life, the goal state can be altered (automatically by a state estimator module or indirectly by a human operator through the features in Section 4.3).

4.5.10 Unknown uncertainty

Requirement

The proposed design must at least result in predictable behavior when detecting non-nominal conditions.

Scope

This thesis proposes a design to counteract this problem.

Approach

Using *state validation* modules, assumptions can continuously be monitored and influence planning on the highest abstract level, directly through state variables in the (augmented) symbolic state or with the guidance of a human expert (see Section 4.3). Through these mechanisms, a predefined envelope of operation can be defined.

5 Experiments

An experiment is performed to validate the proposed architecture. Since not all aspects of the design are implemented, a number of key characteristics are selected for further analysis:

1. Plan creation: is the TFD/M algorithm able to find a solution that satisfies the goal state requirements? Preferably, the proposed plans do not contain large amounts of superfluous movement.
2. Plan monitoring: is new information correctly translated to the symbolic state and replanning triggered when necessary? As new information on corroded points of interest is added to the symbolic state, replanning should be triggered.
3. Plan execution: is the system able to interface with the simulation or hardware, in order to perform an inspection?

The simulated environment used for this experiment can be seen in Figure 5.1. It consists of four compartments, containing a total of three points of interest. Two visual inspection points per compartment are added for the first two compartments. More information on the implementation of the simulator can be found in Appendix B.

In the initial condition, the base is at the beginning of the rail and the arm in an upright position. The clamp is activated, while the lights are off and the Pan-Tilt-Unit is stowed away. No final position for the robot is defined in the goal state, but the arm should be folded, the clamp unactivated, and the lights should be off.

The planner is configured to explore the whole search space without the help of heuristics. When a plan is found, but the search space is not yet fully explored, searching continues for at most 5 seconds.

Video material of the experiment can be found online at http://ceestrouwborst.nl/roboship.

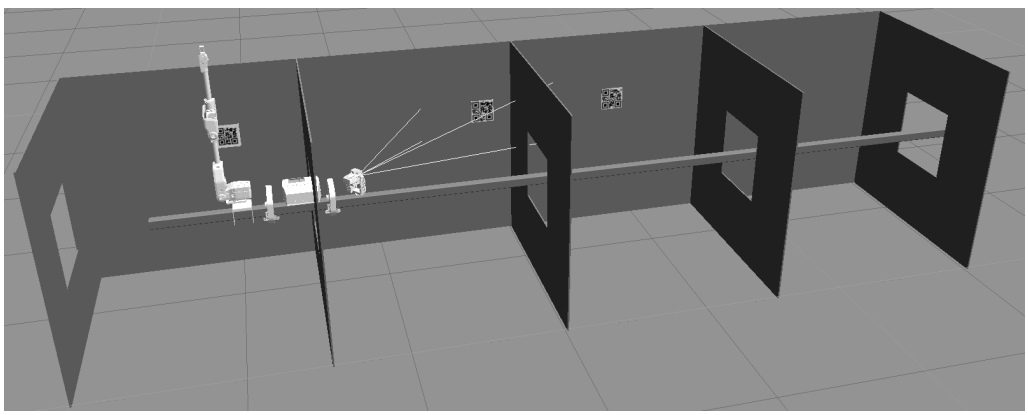


Figure 5.1: View of the simulated environment

6 Results

In this chapter, the results of the experiments are summarized.

Figure 6.3 shows the plan (as in: the sequence of actions as proposed by the discrete planner) over time. Apart from the initial plan, planning is triggered three times, resulting in a total of four plans. The first term between parentheses is the name of the elementary action. The remaining terms are arguments for this specific action. The part of a plan that is not executed, is showed in gray; the final action resulting in satisfied goal state requirements is shown in green.

What cannot directly be seen from the image, but can be deduced from the arguments of the move action, is that no superfluous movement is present in the plan: the robot base moves from the beginning to the end of the rails in one straight line.

Figure 6.1 shows important components of the symbolic state over time. It can be observed that the preconditions of actions are structurally taken into account: arm movement (resulting in an unfolded state) never occurs when the robot base is unclamped, and looking around only happens after unstowing and turning on the lights.[1]

Finally, Figure 6.2 shows the state of the control architecture as a whole. Three key activities are distinguished: creating new plans, monitoring if the current plan still results in the desired goal, and executing actions. The labels refer to the actions in Figure 6.3. Table 6.1 gives numerical information on the balance between the three key activities.

[1] What can be deduced as well is that the lights can be turned off during arm movement, as happens at the end of the plan.

Table 6.1: Time spent in various overall system states

State	Time (s)	% of total time
Planning	21.87	10.32 %
Executing	127.73	60.28 %
Monitoring	62.11	29.31 %
Total	211.89	100 %

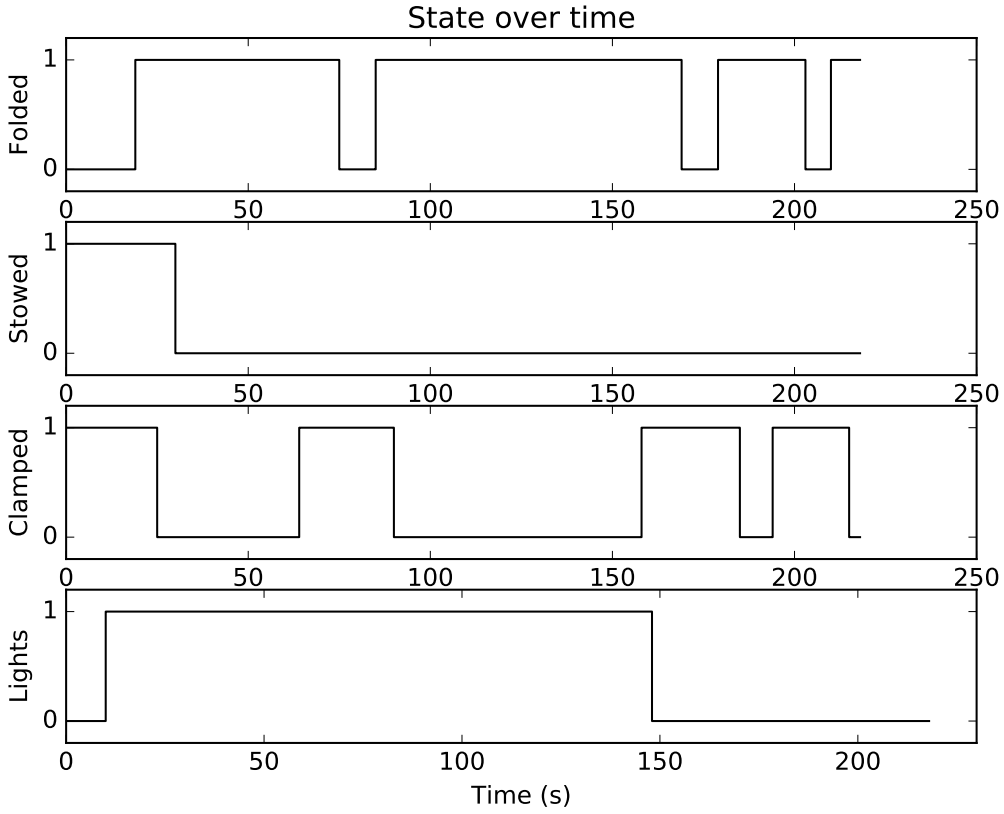


Figure 6.1: Relevant components of the state over time

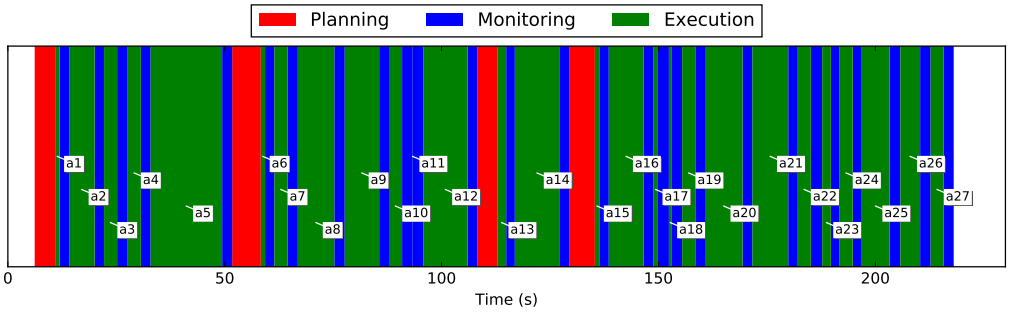


Figure 6.2: Overall system state over time

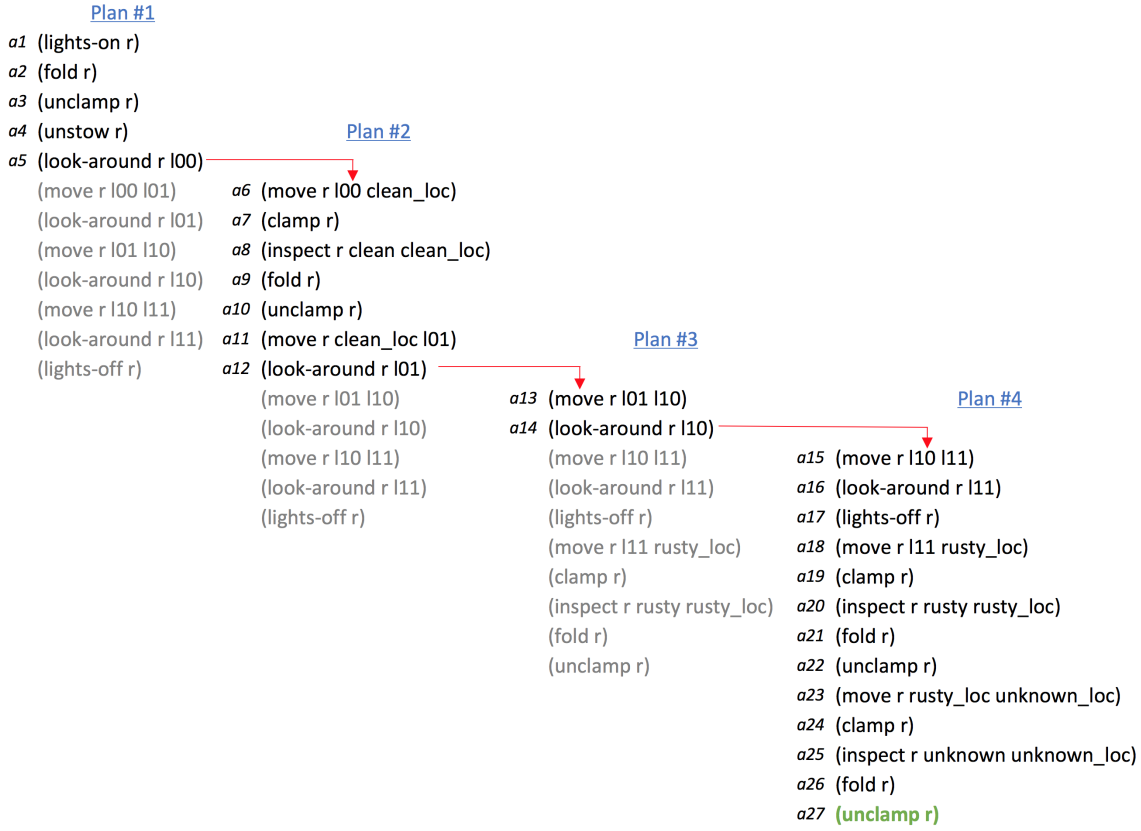


Figure 6.3: Plan and actions over time

7 Evaluation

This chapter reflects on the results shown in the previous chapter. First, the use of the newly introduced simulator will be evaluated. Then, requirements will be evaluated as separate components. Finally, the integration of all components and system level performance will be evaluated.

7.1 Simulator

The newly introduced simulator was found to be very effective and a helpful tool for performing experiments related to high-level autonomous features. Many variations of the same experiment, using a range of parameters for the planner, could be performed in a limited amount of time, without risking damage to the hardware.

Since the dynamics of the system are not of great importance when evaluating high-level autonomous behavior, the assumptions made in the simulator (see Appendix B) do not influence the results and conclusions can be drawn from the experiments in simulation.

7.2 Component level

A subset of the requirements, shown in Table 7.1, can be evaluated using observations from the experiment. The remaining requirements are not implemented in this thesis (annotated with *o* or +) and can therefore not be evaluated.

7.2.1 Geometry

Requirement

The control architecture should be able to perform manipulation planning in a known environment featuring a large number of obstacles.

Evaluation

As described in Chapter 4, manipulator path planning and collision avoidance is implemented using MoveIt!. To validate MoveIt! integration and validate suitability for this use case, a number of end-effector poses were tested throughout the environment. An example of a path that clearly takes the environment into account is shown in Figure 7.1.

7.2.2 Dynamic goal state

Requirement

The planning component must be able to recognize that the current plan is not effective given new information on the goal state, and trigger replanning.

Table 7.1: Different aspects of the problem (as Table 4.1). Underlined components will be evaluated based on the experiment.

Environment-related		**Task-related**		**Platform-related**	
Geometry	++	Dynamic goal state	++	Robust software	++
Localization	+	Visual rust detection	++, o	State estimation	o
Unreliable connectivity	+	Manipulator movement	++	Battery-powered	+
		Unknown uncertainty	+		

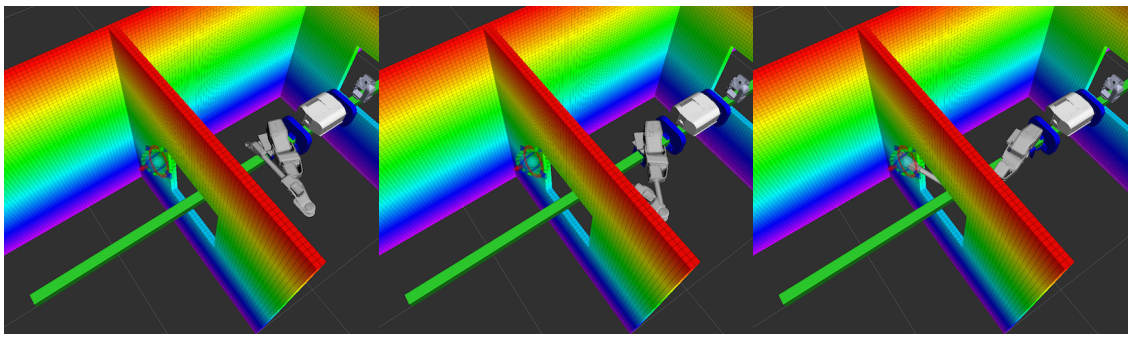


Figure 7.1: Three snapshots of a planned trajectory in a highly obstructed environment

Table 7.2: Change in symbolic state after first *look around* action

Object type	Initial state	After looking around	Comments
rust	None	`clean` (inspected: false) (*x*: 1.004) (*y*: 1.501) (*z*: 1.004)	`clean` is a unique identifier for this point of interest

dist	`l00 l01 l10 l11`
	(seen: false)
		`l00`
		(seen: true)
		`l01 l10 l11`
		(seen: false)
			Point `l00` is used for a look around action.

idist	None	`clean_loc` (*d*: 0.504)	A distance that is to be used to inspect `clean`

Evaluation

To evaluate this requirement, the change in the symbolic state and their properties as result of a *look around* action are examined in Table 7.2. It can clearly be seen that the newly gained knowledge on the environment is indeed added to the state, including all geometric properties.[1] This new knowledge invalidates the plan and results in replanning, as can be seen consistently in Figure 6.3.

7.2.3 Visual rust detection

Requirement

The robotic platform should be able to accurately classify rust and report on the accuracy of its classification.

Evaluation

While visual rust detection is not part of this thesis, the use of raycasting to derive the location of a point of interest is. To evaluate this mechanism, the derived location for the three QR-codes is compared to their actual location in Table 7.3. The discrete nature of raycasting to an OctoMap is clearly visible in the values for the *y*- and *z*-coordinates.

In the current configuration, the error is well under 0.1 meter and deemed sufficiently accurate. If the accuracy needs to be increased, the maximum cell size of the OctoMap can be decreased.

[1] The 0.5 m difference between the *d*-value of `clean_loc` and the *x*-coordinate of `clean` can be explained by the 0.5 m static transformation in *x* between the map- and the rail frames.

Table 7.3: Estimated and real location of detected points of interest (POIs)

POI	**Estimated location** (m) (x, y, z)	**Real location** (m) (x, y, z)	**Error** (m)
1	(0.981, 1.501, 1.004)	(1.000, 1.460, 1.000)	0.045
2	(2.848, 1.501, 1.004)	(2.800, 1.460, 1.000)	0.063
3	(3.840, 1.501, 1.004)	(3.800, 1.460, 1.000)	0.057

7.2.4 Manipulator movement

Requirement

The geometric planner should be able to handle partial setpoints in Cartesian space.

Evaluation

As discussed in Chapter 4, the 5-DOF setpoint is implemented by increasing the tolerance of one of the components of the 6-DOF setpoint. No optimized search algorithm compatible with partial Cartesian setpoints is used. As can be seen in Figure 7.2, this results in the expected behavior for the end effector. During this experiment, the setpoint has zero-valued components for all rotations. While the end effector is placed perpendicular to the environment, the rotation around the axis perpendicular to the wall is not fixed.

7.2.5 Robust software

Requirement

The software interfacing with the different subsystems should be robust, documented and tested.

Evaluation

While the work on software cannot directly be tested using an experiment, it is worth evaluating the structural changes to the software. Three main improvements can be observed, that are discussed in more detail in Appendix E.

- All software packages are examined, cleaned up and at least partly documented.
- All code is now under version control using the Git-based servers provided by the Robotics and Mechatronics group to improve maintainability and traceability.
- Continuous Integration methods are used to validate the correct definition of dependencies and test for compile-time errors.

7.3 System level

In this section, the system level performance is evaluated. As a starting point, the questions posed in Chapter 5 are answered.

1. *On plan creation:* The planner is able to find plans in the domain as described by the PDDL/M file. The plans indeed result in an end state satisfying the requirements given for the goal state.
2. *On plan monitoring:* Replanning is triggered as expected. New information is added to the symbolic state as results of an explicit sensing action, effectively invalidating the plan. The monitoring process recognizes this event, and triggers replanning.
3. *On plan execution:* The interface between the discrete planning domain and the continuous planning- and execution domains is effective. The semantic gap is successfully bridged and an inspection is performed.

An additional number of observations can be made:

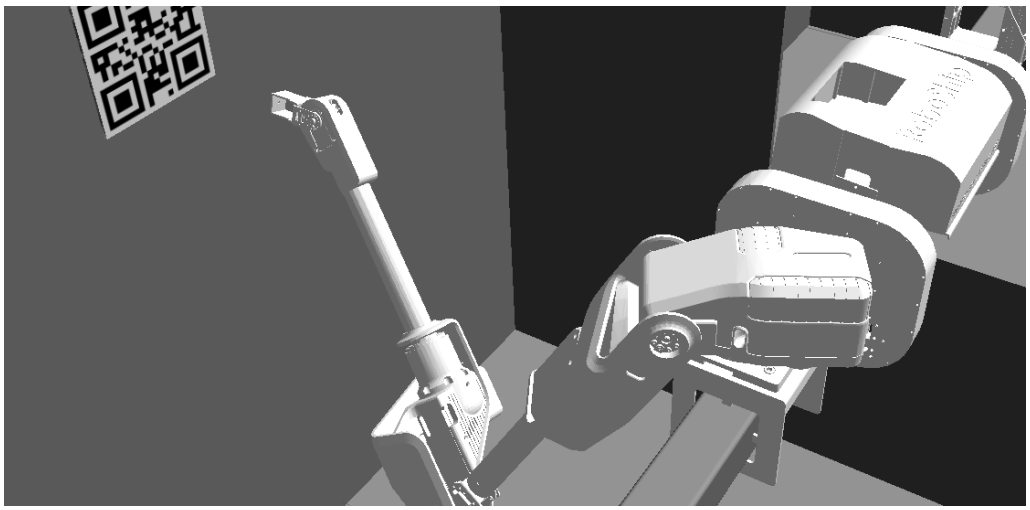


Figure 7.2: Final pose of end-effector before inspection

- The monitoring activity takes a significant amount of the total runtime (almost 30%). While the monitoring activity is key for a successful activity, a less time-consuming monitoring method would be beneficial.
- The planner is not guided by a heuristic during its search. While the total planning time is not significant in this experiment, the search space might become too large to fully explore in a larger environment. A (possibly domain-specific) heuristic would be beneficial in that case. Implementing a heuristic is not a guarantee for an increase in performance, as is shown in Appendix F.

In summary, the experiment shows that the discrete planner at the core of the proposed control architecture, and its interfacing to the continuous domain, can effectively increase the level of autonomy of the robotic inspection platform.

8 Conclusion

As an answer to the research question,

> How to increase the autonomy of the RoboShip platform to the level of *(blended) executive control*, given the fundamental task- and environment uncertainty of the inspection domain?

a layered control architecture was proposed. The key component in the deliberate layer, responsible for high-level autonomous behavior, is a domain-independent discrete planning algorithm based on TFD/M.

This thesis shows that discrete planning systems can be very effective in introducing autonomy to an inspection robot. By altering the TFD/M planner to allow for changes in the goal state requirements as a result of elementary sensing actions, new information on the environment can successfully be taken into account. Using a simulation, it has been shown that this feature in combination with a strict monitoring process results in effective planning and execution of actions.

Additionally, it has been shown that by using a discrete planning system with support for *modules*, the semantic gap can be bridged and relevant details on the continuous domain can be incorporated in the highest, most abstract layer of planning and control.

While not implemented or validated in this thesis, the newly proposed architecture allows for and encourages a supervisory role for the human operator, in which the robot takes initiative in asking help in difficult situations. This explicit interface between human and robot can also be used to create datasets for training classifiers.

The simulator created as part of this research proved to be a very useful tool in researching and validating abstract, high-level control logic.

We believe that the proposed architecture and accompanying tools form a solid foundation for introducing autonomous features to the RoboShip platform, answering to the original research question *How can one transform the RoboShip platform to an autonomous inspection system?*

9 Recommendations and future work

Since this thesis proposes a full control architecture but implements a subset of the proposed features, a large amount of future work can be described. Two large opportunities are related to the further implementation of the proposed design:

- Implementation of the proposed human-robot interface. It is key for handling uncertainty.
- Revising software available for the hardware for interfacing with the planner, enabling tests using the prototype inspection platform.

Besides those two projects, a large number of possible improvements can be described:

- The monitoring process can be examined and simplified, to introduce a significant performance boost.
- A domain-specific heuristic can be introduced, to guarantee a short search time in large environments. Such a heuristic should be chosen with care, since the use of a heuristic can also decrease performance (see Appendix F).
- Information on the changes to the end-effector can be added to the models. The two-stage nature of the manipulator will require additional elementary actions. Modeling contact with the environment will require changes to the simulation and geometric planning framework, as well.
- Additional aspects of deliberation can be implemented, resulting in a “smarter” or more human-like agent. Examples are *excuses and preferences* [Göbelbecker et al. (2010)] and *foresight* [Levihn et al. (2013)].

In the broader RoboShip scope, two additional opportunities can be defined:

- The hardware can be improved to solve the described problems related to state definition.
- A solid localization solution will enable use in real environments.

Bibliography

M. A. Abdelhady. Localization Framework for a Rail-guided Robot. Premsc report 016ram2016, University of Twente, apr 2016.

R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17(4):315–337, 1998.

R. C. Arkin. *Behavior-based robotics*. MIT Press, 1998. ISBN 0262011654.

G. B. Barbieri. Control architecture for docking UAVs with a 7-DOF manipulator. Msc report 045ram2016, University of Twente, 2016.

J. M. Beer, A. D. Fisk, and W. A. Rogers. Toward a Framework for Levels of Robot Autonomy in Human-Robot Interaction. *Journal of Human-Robot Interaction*, 3(2):74, 2014. ISSN 2163-0364. doi: 10.5898/JHRI.3.2.Beer.

D. J. Borgerink, J. Stegenga, D. M. Brouwer, H. J. Wortche, and S. Stramigioli. Rail-guided robotic end-effector position error due to rail compliance and ship motion. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3463–3468, 2014. ISBN 2153-0858. doi: 10.1109/IROS.2014.6943045.

D. J. Borgerink, D. M. Brouwer, J. Stegenga, and S. Stramigioli. Kinematic Design Method for Rail-Guided Robotic Arms. *Journal of Mechanisms and Robotics*, 9(1), 2016. ISSN 1942-4302. doi: 10.1115/1.4035187.

J. F. Broenink and Y. Ni. Model-driven robot-software design using integrated models and co-simulation. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 339–344. IEEE, jul 2012. ISBN 978-1-4673-2297-3. doi: 10.1109/SAMOS.2012.6404197.

R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1):14–23, 1986. ISSN 0882-4967. doi: 10.1109/JRA.1986.1087032.

L. Christensen, N. Fischer, S. Kroffke, J. Lemburg, and R. Ahlers. Cost-Effective Autonomous Robots for Ballast Water Tank Inspection. *Journal of Ship Production & Design*, 27(3):127–136, 2011. ISSN 00811661.

J. Connell. SSS: a hybrid architecture applied to robot navigation. *Proceedings 1992 IEEE International Conference on Robotics and Automation*, pages 2719–2724, 1992. doi: 10.1109/ROBOT.1992.219995.

G. P. S. de Carvalho. *Localization Of An Autonomous Rail-Guided Robot Based On Particle Filter*. PhD thesis, Alberto Luiz Coimbra Institute of Post-Graduation and Research in Engineering, 2016.

C. Dornhege. *Task Planning for High-Level Robot Control*. PhD thesis, University of Freiburg, 2015.

C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel. Semantic Attachments for Domain-Independent Planning Systems. *Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

E. Erdem, V. Patoglu, and P. Schüller. A Systematic Analysis of Levels of Integration between Low-Level Reasoning and Task Planning. *AI Communications*, 2016. ISSN 09217126. doi: 10.3233/AIC-150697.

P. Eyerich, R. Mattmüller, and G. Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. *Springer Tracts in Advanced Robotics*, 76:49–64, 2012. ISSN 16107438. doi: 10.1007/978-3-642-25116-0_6.

R. Firby. An investigation into reactive planning in complex domains. *AAAI*, 19751:202–206, 1987. ISSN 1543-5474. doi: 10.1016/0007-6813(87)90018-8.

S. Franklin and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Intelligent agents III agent theories, architectures, and languages*, pages 21–35. Springer, Berlin, Heidelberg, 1997. ISBN 3540625070. doi: 10.1007/BFb0013570.

C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling. Backward-forward search for manipulation planning. *IEEE International Conference on Intelligent Robots and Systems*, 2015-Decem (grant 1420927):6366–6373, 2015. ISSN 21530866. doi: 10.1109/IROS.2015.7354287.

E. Gat. On Three-Layer Architectures. *Artificial intelligence and mobile robots*, pages 195–210, 1997. doi: 10.1.1.165.5283.

M. Ghallab, D. S. Nau, and P. Traverso. *Automated planning and acting*. 2016. ISBN 9781107037274.

S. H. Gies. Mapping and Localization of a Rail-guided Robot. Bsc report 019ram2015, University of Twente, 2015.

M. Göbelbecker, T. Keller, and P. Eyerich. Coming up with good excuses: What to do when no plan can be found. *Proceedings 2010 International Conference on Automated Planning and Scheduling*, (Icaps):81–88, 2010.

J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: an efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, apr 2013. ISSN 0929-5593. doi: 10.1007/s10514-012-9321-0.

J. A. J. Huttenhuis. Kinematic Design Method for a Rail Mounted Inspection Robot Arm. Master's thesis, University of Twente, 2015.

L. Kaelbling and T. Lozano-Perez. Integrated robot task and motion planning in the now. Technical report, MIT CSAIL, 2012. URL http://hdl.handle.net/1721.1/71521.

L. P. Kaelbling and T. Lozano-Pérez. Integrated Task and Motion Planning in Belief Space. *International Journal of Robotics Research*, 32, 2013.

S. M. LaValle. *Planning Algorithms*. 2006. ISBN 9780511546877. doi: 10.1017/CBO9780511546877.

M. Levihn, L. P. Kaelbling, T. Lozano-Perez, and M. Stilman. Foresight and reconsideration in hierarchical planning and execution. *IEEE International Conference on Intelligent Robots and Systems*, pages 224–231, 2013. ISSN 21530858. doi: 10.1109/IROS.2013.6696357.

R. Murphy. *Introduction to AI robotics*. MIT Press, 2000. ISBN 9780262133838.

N. J. Nilsson. Shakey The Robot. Technical Report 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, apr 1984.

L. M. Overbeek. Controller Design for a Rail Mounted Inspection Robot Manipulator. Msc report 034ram2015, University of Twente, nov 2015.

E. H. Rijnbeek. Rail-guided ballast tank inspection robot, Hardware infrastructure analysis and enhancement. Bsc report 020ram2015, University of Twente, 2015.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 3rd edition*. 2009. ISBN 9780136042594. doi: 10.1017/S0269888900007724.

SmartBot. RoboShip website, 2016. URL http://www.smartbot.eu/nl/roboship/.

S. Srivastava, L. Riano, S. Russell, and P. Abbeel. Using Classical Planners for Tasks with Continuous Operators in Robotics. *ICAPS Workshop on Planning and Robotics*, 2013.

S. Stramigioli, E. D. Fasse, and J. C. Willems. A rigorous framework for interactive robot control. *International Journal of Control*, 75(18):1486–1503, jan 2002. ISSN 0020-7179. doi: 10.1080/00207170210000320078.

M. Ten Den. Design of a 3 DOF end-effector for wall inspection. Technical report, University of Twente, 2016.

S. Thrun, W. Burgard, and D. Fox. *Probabilistic robotics*. 2005. ISBN 9780262201629. doi: 10.1145/504729.504754.

S. Trüg. An integration of manipulation and action planning. Technical report, University of Freiburg, 2006. URL http://gki.informatik.uni-freiburg.de/papers/diplomarbeiten/trueg-diplomarbeit-06.pdf.

A. Van Breemen and T. De Vries. Design and implementation of a room thermostat using an agent-based approach. *Control Engineering Practice*, 9(3):233–248, mar 2001. ISSN 09670661. doi: 10.1016/S0967-0661(00)00111-8.

Appendices

A Localization

To understand the challenge of localization to the fullest, and propose a meaningful solution for future work, a significant amount of time was invested in the design and implementation of a localization solution. Two project were performed on the localization of the RoboShip platform. One solution relied on dead-reckoning within a pre-defined map [Abdelhady (2016)], but the amount of slip present in the system was too large to provide an accurate estimate. The other solution proposed an Extend Kalman Filter, that was at the center of both mapping and localization. Information on the different parts the rail was made of (bends and straight pieces) were used to create a map that could be used for localization as well. Detecting the bends turned out to be difficult, however, resulting in an inaccurate map and localization estimate [Gies (2015)].

Based on the conclusions from the previous projects, successful implementations in similar situations ([Christensen et al. (2011); de Carvalho (2016)]), and the excellent overview of available approaches given by [Thrun et al. (2005)], this thesis proposes to implement a particle filter that is used with a pre-defined map. In general, CAD data on the tanks is available, allowing the creation of accurate maps beforehand.

Specifically, an Augmented Monte Carlo Localization scheme is proposed, which is a particle filter with an explicit motion and measurement model based on the used robot [Thrun et al. (2005)]. The motion model is based on wheel encoder data, while the measurement model incorporates RFID readings and orientations measurements as known landmarks. The “augmented” (or “adaptive”) term indicates that random particles are inserted to add an additional level of robustness: the localization algorithm will be able to recover from catastrophic failures and can handle the “kidnapped robot” problem (a sudden, usually undetectable change in location).

The filter, apart from the random sample insertion, was implemented in Python as a one-dimensional filter. A tool was created to show the current state of the filter and visualize the landmarks present in the map (Figure A.1). The translation from 1D to 3D is performed based on the map of the tank, and was implemented as well. This is non-trivial, since the different carts of the robot are connected through universal joints. For each of the carts, a separate 1D to 3D transformation needs to be performed, based on the 1D location of that specific cart on the rail.

At this point, the flexibility of the rail is not taken into account. If future work concludes that this aspect is significant, it is recommended to extend the 1D to 3D conversion based on a model of the rail and information from the camera. This allows for both a relatively simple filter and high accuracy end-effector localization.

While a large part of the filter is implemented, it remains largely untested. The main reason for this is that no obivious methods for creating a “ground truth” were available to test the performance of the filter. Additional remarks on the current implementation:

- In order to use the Inertial Measurement Unit in the particle filter, large changes in orientation are detected (effectively the bends in the rail) and orientation measurement at that time is compared to orientation landmarks (orientation estimates in all bends). While this might work in theory, the stability and accuracy of the orientation measurements are unknown. Replacing this by an approach based on the measured magnetic field might be beneficial and more robust (as proposed originally in [Christensen et al. (2011)]). Both methods require a manually operated run to create the map.

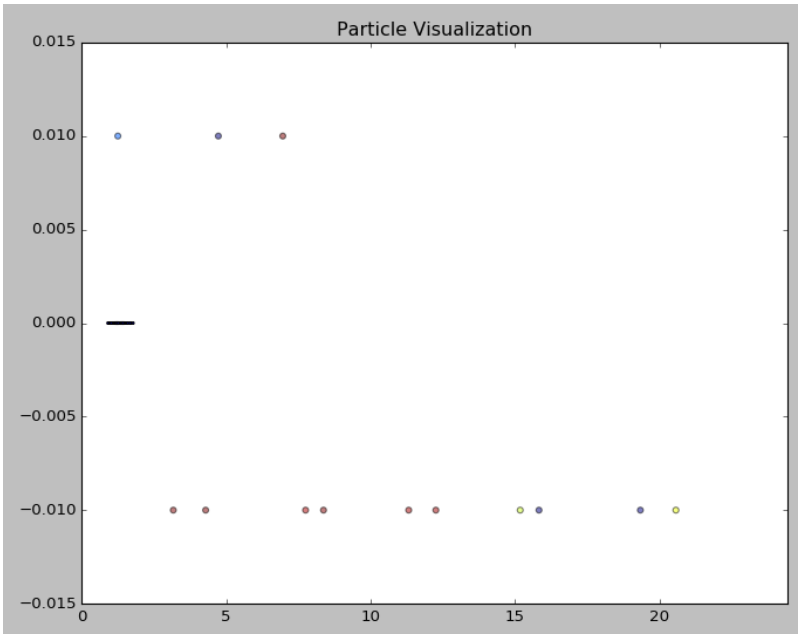


Figure A.1: Introspection tool for the visualization of particles and landmarks. Particles are shown at the x-axis. Orientation landmarks are shown at $y = -0.01$, while RFID landmarks are shown at $y = 0.01$.

- The sensor- and motion models are not tuned and tested. Normal distributions are used with a constant, pre-defined standard deviation. To what extent these probabilities represent the real system, is unknown.
- To get to a final pose estimate, a Gaussian mixture model is fitted to the particle density. To find the number of required components, the Bayesian Information Criterion is used. The component with the smallest covariance is used to select the definite value for the pose estimate. This approach is not tested.

B Simulation

Working in simulation has a number of advantages over working with a real robotic platform during development. For the RoboShip platform, the following aspects were key drivers in the choice for introducing a simulator:

- No solid localization component is available for the real robotic platform, increasing the risk of collisions that might possibly damage the robot.
- Not all subsystems of the robot are thoroughly tested, increasing the risk of unpredictable events that might damage the robot.
- Performing a large number of experiments with the real robotic platform is impractical, seems it battery-powered and not stored near the experimental tank.

To create the simulator, Gazebo[1] was used. It integrates very well with ROS, and is therefore the de facto standard in the ROS community. All meshes used in the simulation, for both the robot and the environment, are derived from CAD files created in SolidWorks or OnShape. The simulator can be seen in Figure B.1.

The following sections will provide details on the implementation of the simulation for the arm and base. The last section of this appendix will provide important information on the assumptions on the geometry of the environment.

Arm implementation

In reality, the actual movement of the arm depends a lot on the performance of the Dynamixel servo motors that are used. Besides that, the flexibility of the links contribute significantly to the total positioning error. Those aspects of the manipulator are not modeled in the simulation, by using rigid links and “position joint interfaces” for the actuators. When a setpoint is given to the joint interface, the joint will directly be at the desired position, independent of the mass of the links or other dynamic effects.

Base implementation

Simulating the interaction between the drive units and the rail is complicated. In reality, the drive units are clamped onto the rail using a number of very stiff springs. This, combined with the elasticity of the wheels themselves, results in a stable fix on the rails. The control of the drive units does not take into account bends in the rail and thus relies on some slip in the wheels for going around corners. While very interesting, the choice was made not to

[1] http://gazebosim.org/

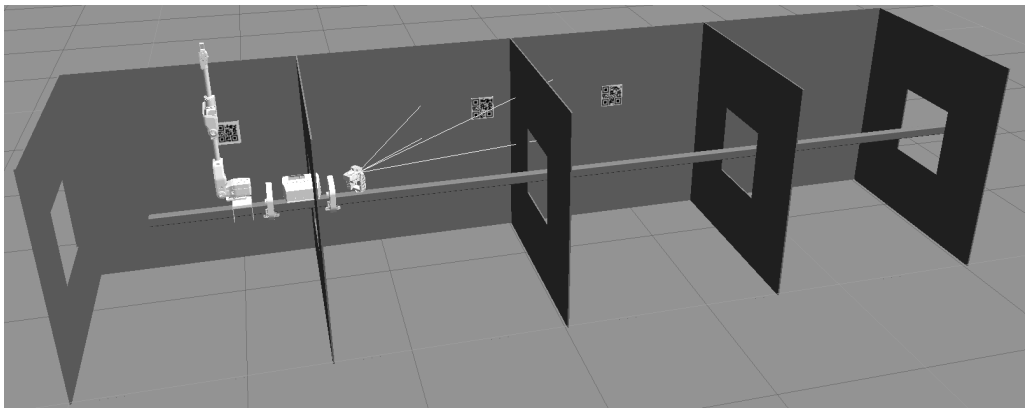


Figure B.1: Snapshot of the simulator

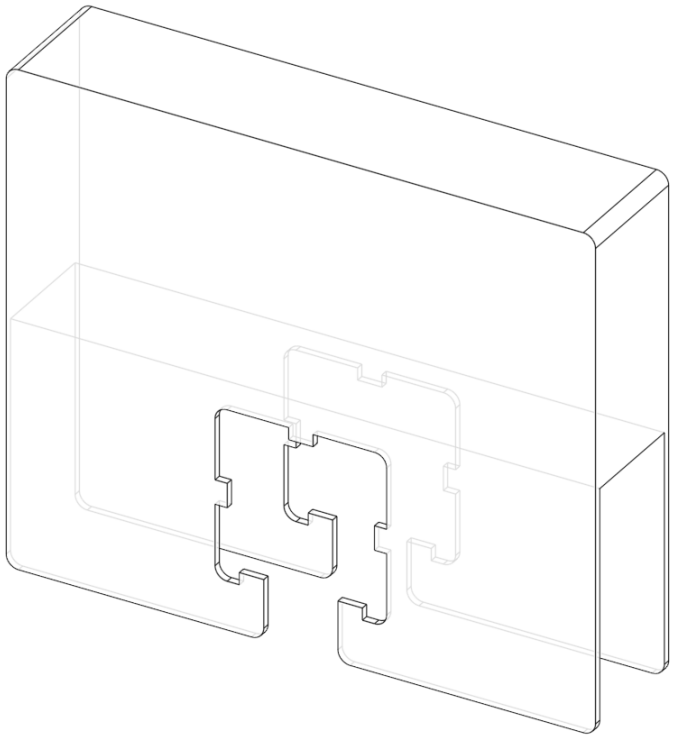


Figure B.2: Altered collision mesh for the drive-units

simulate this part of the dynamics. The interaction between the rails and the robot is modelled as a sliding contact. To model the sliding contact in a way that allows going around corners, which might be desirable in the future, the meshes used for collisions are different from the visualization meshes (Figure B.2). While the sliding contact does work sufficiently well, it should be mentioned that it requires a significant amount of resources, and results in some vibrations as well.

A second point of interest is the interconnection between the carts that form the base. To model the universal joints used in the mechanical construction, a combination of three rotational joints is used per joint. While multi-DOF joints are supported by Gazebo, not all parts in the software stack (MoveIt! specifically) handle them correctly.

Geometry

The geometry of the rail is greatly simplified in the simulation: no bend are present and the orientation of the robot is constant. The main reason for this is that, as explained in the previous section, locomotion is not explicitly modeled. To make the robot move, its pose is increased with small increments, based on a description of the rail present in the map and a description of the robot geometry. The simulation is untested for more realistic, complex rail shapes.

Conclusions and future work

The simulator proved a very useful tool in this thesis on high-level control. It is recommended to extend the simulator for autonomy experiments in more challenging environments, equivalent to the experimental tank. Given the current hardware, it does not seem worthwile to simulate the dynamics more accurately.

C Arm Control

Control of the manipulator of the RoboShip platform was researched in two earlier projects [Huttenhuis (2015); Overbeek (2015)]. The approach taken for manipulator control was transpose Jacobian control (for reference, see Stramigioli et al. (2002)). To implement this kind of control on the Dynamixel servo motors that require position setpoints, the behavior of the arm was modeled and the expected joint poses used as setpoints for the actual controllers. All implementation was done in MATLAB.

Integrating the results of the previous projects in this thesis would require multiple steps:

- First, the code would need to be ported to Python or C++ and tested.[1]
- Then, existing path planning and colission avoidance approaches should be researched and interfaced with the arm controller.
- A low-level interface between the arm model and the Dynamixel motors in the joints would need to be created, to use the calculated state as setpoints.

Since arm control is not at the core of this thesis, a different solution was chosen that results in similar results: the use of MoveIt! for ROS[2]. MoveIt! streamlines the process of modeling and controlling a robotic manipulator, includes tools for visualizations and provides interfaces to a number of geometric planning algorithms. Obstacle avoidance is provided out-of-the-box, and the software proved effective in other projects at the Robotics and Mechatronics group (e.g. Barbieri (2016)).

Some details on the implementation of MoveIt! are listed below. They require some knowledge on the inner workings of MoveIt!, and are mainly included for future reference.

- The whole robot (all base modules and the manipulator) is described in one URDF file, to be able to check for self-colissions. This URDF file is also used directly in the simulator, so includes information on colission meshes and sensors.
- MoveIt! is used in a relatively standard configuration, that uses a path planning algorithm based on Rapidly-Expanding Random Trees.
- The rail is added to the planning scene as a collision object using the mesh file directly. The tank is first converted to an OctoMap [Hornung et al. (2013)], to allow for raycasting during the *look around* action. The discretization slightly, but not significantly, reduces accuracy. A visualization of the environment as known by the robot can be seen in Figure C.1.
- Interfacing to the actual Dynamixel servo motors is done through the `dynamixel_motor` package[3].
- To keep the environment relatively static, the whole map is used as basis for the planning scene. The transform from the map frame to the robot is implemented as a virtual joint in the SRDF file.

[1] Technically, ROS can also interface with MATLAB, but introducing a dependency on a large, expensive, closed-source software package should be avoided.

[2] http://moveit.ros.org/

[3] http://wiki.ros.org/dynamixel_motor

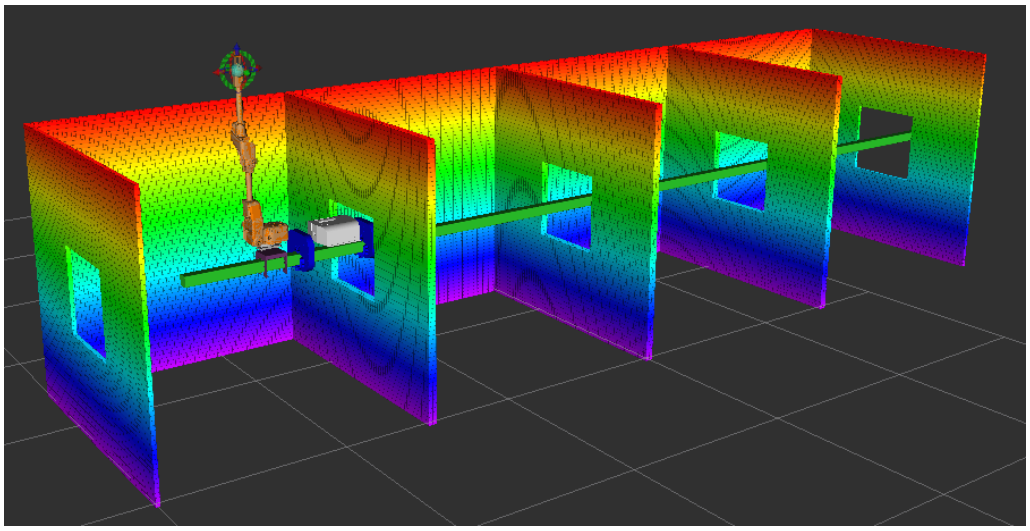


Figure C.1: Visualization of the environment as perceived by the robot, and used for manipulator path planning.

D PDDL/M domain description

This appendix shows the PDDL/M domain description file, containing all elementary actions, types and predicates. The required modules are listed as well.

```
(define (domain rs-planning)
    (:requirements :strips :typing :durative-actions :fluents :derived-predicates :equality :conditional-effects :modules)

    (:types
        robot                           ; The inspection platform

        dist                           ; A 1D distance from the beginning of the rail
        idist - dist                    ; A dist specifically for the inspection of a nearby pose

        pose                            ; any pose in space, assumed in the global map frame.
        rust - pose                     ; Rust is nothing more (at this point) than a pose, but represents rust.
    )

    (:modules
            (moveCosts ?r - robot ?s - dist ?g - dist
                                cost moveCosts@librs_planning_mods.so)
            (reachable ?r - robot ?p - rust ?d - dist
                            conditionchecker reachable@librs_planning_mods.so)
    )

    (:predicates
        (stowed ?r - robot)              ; Is the camera of robot r stowed?
        (clamped ?r - robot)                    ; Is the clamp of robot r activated?
        (lights ?r - robot)                     ; Are the lamps of robot r on?
        (folded ?r - robot)                     ; Is the arm of robot r folded in?
        (seen ?d - dist)                        ; Is rail distance d used for a look_around?
        (inspected ?r - rust)                   ; Is rusty spot r inspected?
        (at-location ?r - robot ?d - dist)      ; Is the robot r at rail distance d?
    )

    (:functions
        ; Position
        (x ?p - pose) - number
        (y ?p - pose) - number
        (z ?p - pose) - number
        ; quaternion orientation
        (qx ?p - pose) - number
        (qy ?p - pose) - number
        (qz ?p - pose) - number
        (qw ?p - pose) - number
        ; Rail location distance
        (d ?d - dist) - number
    )

    (:durative-action lights-on
        :parameters (?r - robot)
        :duration (= ?duration 1.0)
        :condition
        (and
            (at start (not (lights ?r)))
        )
        :effect
        (and
            (at end (lights ?r))
        )
    )

    (:durative-action lights-off
        :parameters (?r - robot)
        :duration (= ?duration 1.0)
        :condition
        (and
            (at start (lights ?r))
        )
        :effect
        (and
            (at end (not (lights ?r)))
        )
    )

    (:durative-action clamp
        :parameters (?r - robot)
        :duration (= ?duration 1.0)
        :condition
        (and
            (at start (not (clamped ?r)))
        )
        :effect
        (and
            (at end (clamped ?r))
        )
    )

    (:durative-action unclamp
        :parameters (?r - robot)
        :duration (= ?duration 3.0)
        :condition
        (and
            (at start (clamped ?r))
```

```
        )
        :effect
        (and
            (at end (not (clamped ?r)))
        )
    )

    (:durative-action stow
        :parameters (?r - robot)
        :duration (= ?duration 1.0)
        :condition
        (and
            (at start (not (stowed ?r)))
        )
        :effect
        (and
            (at end (stowed ?r))
        )
    )

    (:durative-action unstow
        :parameters (?r - robot)
        :duration (= ?duration 3.0)
        :condition
        (and
            (at start (stowed ?r))
        )
        :effect
        (and
            (at end (not (stowed ?r)))
        )
    )

    (:durative-action fold
        :parameters (?r - robot)
        :duration (= ?duration 8.0)
        :condition
        (and
            (at start (clamped ?r))
            (at start (not (folded ?r)))
        )
        :effect
        (and
            (at end (folded ?r))
        )
    )

    (:durative-action look-around
        :parameters (?r - robot ?d - dist)
        :duration (= ?duration 4.0)
        :condition
        (and
            (at start (lights ?r))
            (at start (not (stowed ?r)))
            (at start (folded ?r))
            (at start (at-location ?r ?d))
        )
        :effect
        (and
            (at end (seen ?d))
        )
    )

    (:durative-action inspect
        :parameters (?r - robot ?p - rust ?d - idist)
        :duration (= ?duration 10.0)
        :condition
        (and
            (at start (at-location ?r ?d))
            (at start (clamped ?r))
            (at start ([reachable ?r ?p ?d]))
        )
        :effect
        (and
            (at end (inspected ?p))
            (at end (not (folded ?r)))
        )
    )

    (:durative-action move
        :parameters (?r - robot ?s - dist ?g - dist)
        :duration (= ?duration [moveCosts ?r ?s ?g])
        :condition
        (and
            (at start (at-location ?r ?s))
            (at start (folded ?r))
            (at start (not (clamped ?r)))
        )
        :effect
        (and
            (at end (at-location ?r ?g))
            (at end (not (at-location ?r ?s)))
        )
    )
)
```

E Software implementation

During this thesis, the codebase for the RoboShip project was expanded and restructured significantly. This appendix describes the software structure at the end of the thesis.

ROS

ROS, short for *Robot Operating System*, is used as software framework in the RoboShip project. It is great for prototyping robotics software, mainly because the large number of available open packages and the very active community (which can be reached easily through Q&A site ROS Answers[1] and discussion forum ROS Discourse[2]). While ROS was already used at the beginning of the thesis, not one of the original packages was left untouched.

Table E.1 lists the ROS packages at the end of the thesis. In the column *State*, *Cleaned-up* means that the module was renamed, comments were added and basic functionality was tested. *Extended* means that the functionality present at the start of the thesis is still present, but additional features were implemented. Finally, *New* means that the package and its functionality were introduced during this thesis.

Languages used for development are mainly C++ and Python.

[1] http://answers.ros.org/
[2] https://discourse.ros.org

Table E.1: Overview of RoboShip related packages

Name	State	Description	Key dependencies
elcosensor	Cleaned-up	Contains the interface to the Elcosensor inductive sensor	
fake_moving_platorm	New	Contains a very simple "robot" and environment for planning experiments.	
rfid_reader	Cleaned-up	Contains the interface to the RFID reader.	`libnfc`
roboship	New	RoboShip *metapackage*	
rqt_roboship	New	Currently unused code for RoboShip specific RQT-style GUI nodelets.	Qt
rs_arm	Extended	Contains the configuration files for the dynamixel_motor package, and a currently unused and untested Python implementation of [Overbeek (2015)].	`dynamixel_motor`
rs_base	Cleaned-up	Contains the ROS node interfacing with the drive units.	
rs_battery_indicator	Cleaned-up	Contains the ROS node interfacing with the battery monitoring system.	
rs_clamp	Cleaned-up	Contains the ROS node interfacing with the clamp mechanism.	
rs_inputs	New	Contains nodes for using a Joystick or SpaceMouse to interact with various part of the RoboShip platform.	
rs_led_module	Cleaned-up	Contains the ROS node interfacing with the LED module connected to the camera	
rs_localization	New	Contains the localization functionality described in Chapter A.	
rs_messages	New	Contains the message types and interfaces used throughout the system.	
rs_moveit	New	Contains a number of helper nodes for working with MoveIt!, most importantly helpers for loading the collision environment.	
rs_planning	New	Contains the RoboShip specific action executors, goal creators, state creators and modules for the TFD/M planning system.	`tfd_modules`
rs_ptu	New	Contains the configuration files for the dynamixel_motor package, specific for the Pan-Tilt-Unit. Also implements the *look around* action server.	`dynamixel_motor`
rs_raycast	New	Implements the raycasting function to derive the position of rust from the camera pose and environment.	MoveIt!, `octomap_ros`
rs_robot_description	New	Contains URDF-, SRDF- and custom YAML files describing the robot	
rs_scenarios	New	Contains convenient launch files for scenarios (for demos)	
rs_simulation	New	Contains the simulation functionality described in Appendix B	Gazebo Simulator
rs_state_machines	New	Contains a (currently unused) state machine for a subset of the RoboShip functionality.	`smach`

Version control and testing

All code is versioned using Git, and can be found on the GitLab server of the Robotics and Mechatronics group.[3] Travis-CI is used for testing dependencies: on every push, a clean Ubuntu 16.04 image is deployed on a virtual server. All dependencies are resolved automatically and all C++ code is compiled. While very useful for early detection of C++ related problems, it should be mentioned that Python problems are not detected this way since they occur at runtime.

[3] https://git.ram.ewi.utwente.nl/

F Planning using domain-independent heuristic

To show the impact of a poorly selected heuristic on the total system performance, an additional experiment was performed. The initial state and goal state requirements are equal to the experiment described in Chapter 5, but a generic domain-independent heuristic is used without further configuration. Specifically, the `cyclic_cg` that is part of TFD/M is used. While not fully defined in the work of C. Dornhege (Dornhege (2015)), it seems to be a heuristic based on *causal graphs*.

The used performance measures are the same as in Chapter 6 and Chapter 7, to allow a direct comparison: All actions performed can be seen as part of the evolution of the plan over time, in Figure F.3. Figure F.1 shows important components of the symbolic state over time. Figure F.2 shows the state of the control architecture as a whole. Table F.1 gives numerical information on the balance between the three key activities.

The following conclusions can be drawn:

- A significant amount of unnecessary movement is performed. Besides that, unnecessary actions are executed as well. Clear examples of both are denoted in red in Figure F.3.
- Using a heuristic does not necessarily result in faster planning: relatively to the situation in which no heuristic is used, a larger percentage of time is spent on planning (17.91% versus 10.32%).
- The total performance can significantly decrease when using a heuristic. The total time spent on the execution of actions takes an additional 12.64 seconds when using a heurist, which is approximately 10% of the time spent on planning in the original experiment.

Table F.1: Time spent in various overall system states (with heuristic)

State	Time (s)	% of total time
Planning	46.78	17.91 %
Executing	140.37	53.73 %
Monitoring	73.60	28.18 %
Total	261.23	100 %

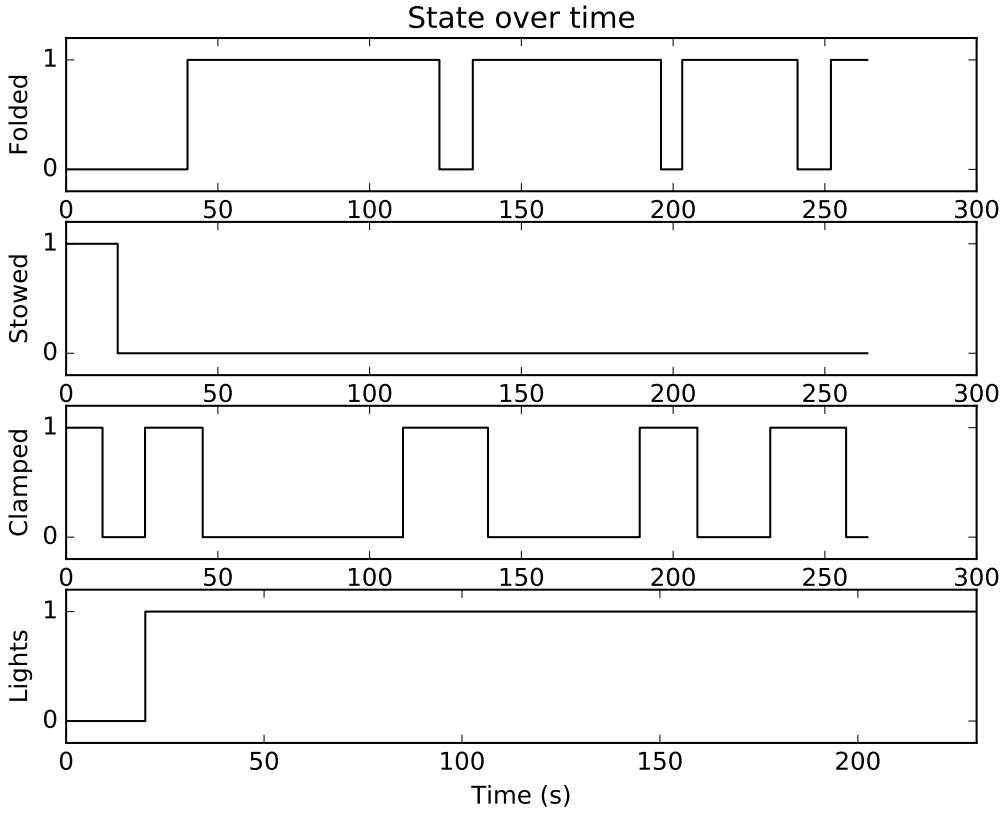


Figure F.1: Relevant components of the state over time (with heuristic)

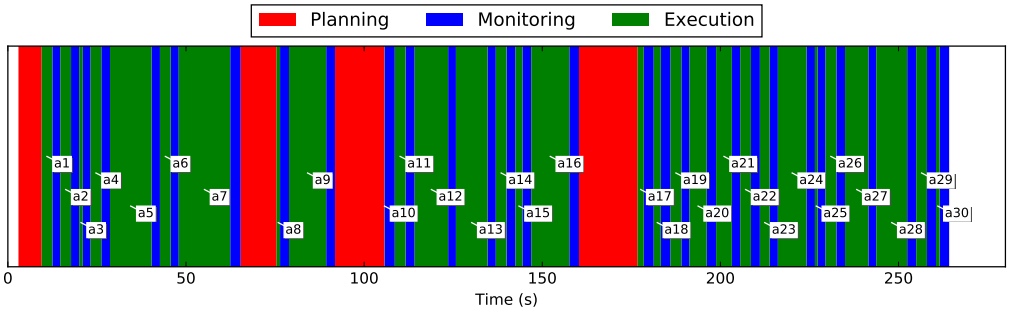


Figure F.2: Overall system state over time (with heuristic)

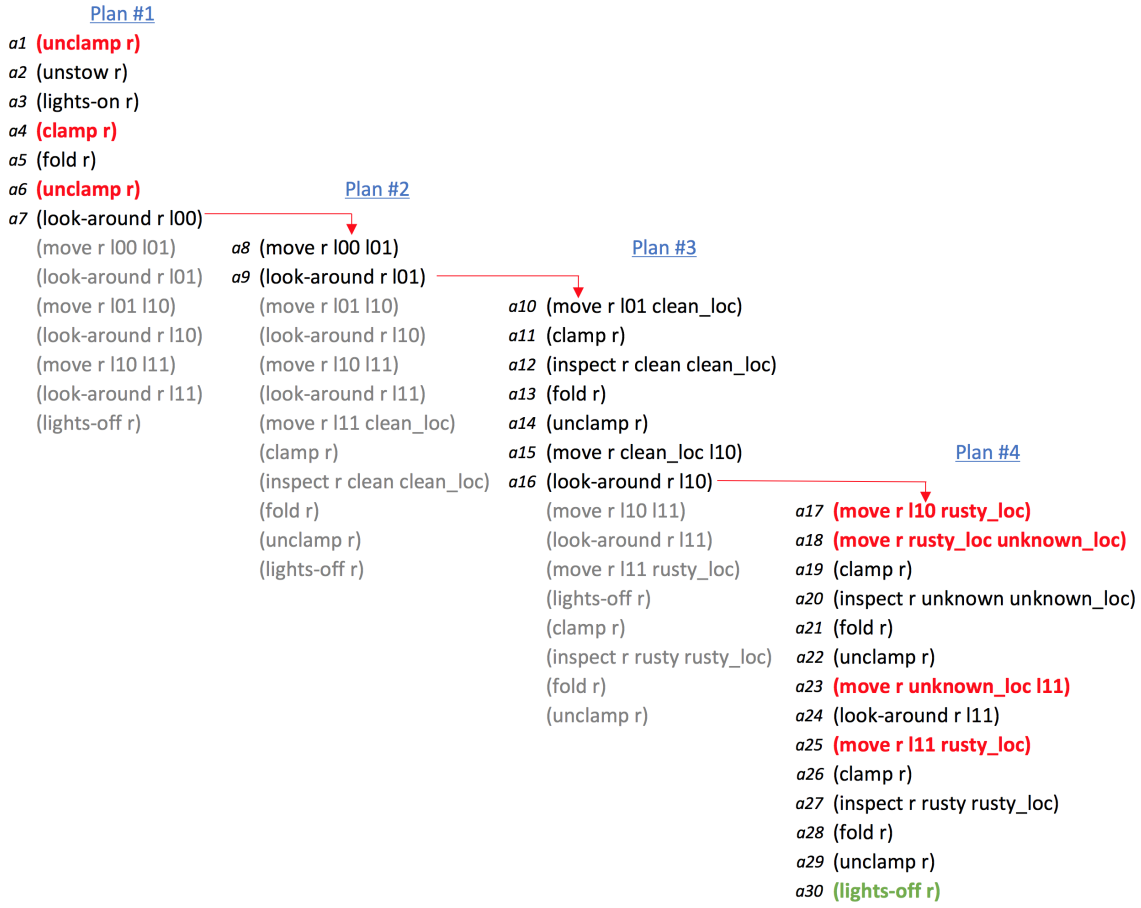


Figure F.3: Plan and actions over time (with heuristic)