Development of a User Interface and Back-End for Planning Tasks using Autonomous Robots

Entwicklung einer Benutzerschnittstelle zur Planung komplexer Inspektionsaufgaben mit autonomen Robotern

Bachelor-Thesis von Stefan Manuel Fabian

Tag der Einreichung:

1. Gutachten: Prof. Dr. Oskar von Stryk
2. Gutachten: Dr.-Ing. Stefan Kohlbrecher

TECHNISCHE UNIVERSITÄT DARMSTADT

Development of a User Interface and Back-End for Planning Tasks using Autonomous Robots
Entwicklung einer Benutzerschnittstelle zur Planung komplexer Inspektionsaufgaben mit autonomen Robotern

Vorgelegte Bachelor-Thesis von Stefan Manuel Fabian

1. Gutachten: Prof. Dr. Oskar von Stryk
2. Gutachten: Dr.-Ing. Stefan Kohlbrecher

Tag der Einreichung:

Abstract

The ARGOS challenge is a competition hosted by TOTAL in collaboration with the ANR (Agence nationale de la recherche) with the objective to create a robot system that can be used for inspection tasks on off- and onshore oil and gas production sites.
The 3rd competition requires an operator without detailed knowledge about the used robot control software to create mission plans after a short introduction.
Currently, the mission plan is directly modeled as a finite state machine using the FlexBe behavior modeling framework.
This approach requires the robot operator to have significant expert knowledge, however.
The user interface presented in this thesis allows for a quick creation of missions with few clicks in a virtual model of the area of robot operation which greatly improves simplicity.
As part of this thesis, aside from the planning user interface, a general purpose overlay for the robot visualization tool *RViz* and an intermediate language for the specification of mission plans have been developed.

Zusammenfassung

Die ARGOS Challenge ist ein Wettbewerb, veranstaltet von TOTAL in Zusammenarbeit mit der ANR (Agence nationale de la recherche). Sie hat zum Ziel, ein Robotersystem zu entwickeln, das Inspektionsaufgaben auf Öl- und Gasförderungsplattformen durchführen kann.
Im dritten Wettkampf soll ein Operator ohne detailliertes Vorwissen über die benutzte Roboterkontrollsoftware nach einer kurzen Einführung Missionspläne erstellen.
Aktuell werden Missionspläne mithilfe des FlexBe Verhaltensmodellierungsframeworks direkt als endlicher Automat modelliert.
Dieser Ansatz erfordert signifikantes Expertenwissen von dem Operator.
Die Benutzeroberfläche, die in dieser Thesis vorgestellt wird, erlaubt ein schnelles und einfaches Erstellen von Missionsplänen in einem virtuellen Modell der Einsatzumgebung.
Als Teil dieser Thesis wurde neben der Benutzeroberfläche ein universelles Overlay für das Robotervisualisierungstool *RViz* und eine Zwischensprache für die Spezifizierung von Missionsplänen entwickelt.

Contents

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. ARGOS Challenge . . . 2
- 1.3. Robot Operating System . . . 5
- 1.4. RViz . . . 6
- 1.5. Ogre . . . 7
- 1.6. Qt . . . 8

2. State of Research **9**

3. Concepts **10**
- 3.1. Visual Representation of the Mission . . . 10
- 3.2. Internal Representation of the Mission . . . 11
- 3.3. User Interface . . . 13
- 3.4. Design . . . 15

4. Software **16**
- 4.1. Implementation . . . 16
- 4.2. General Displays, Tools and View Controllers . . . 16
  - 4.2.1. Mouse View Controller . . . 16
  - 4.2.2. Tool Selection Display . . . 17
  - 4.2.3. UMAD Floor Control . . . 17
  - 4.2.4. Robot Navigation Tool . . . 17
  - 4.2.5. Place Obstacle Tool . . . 18
- 4.3. Overlay . . . 19
  - 4.3.1. Rendering the Overlay . . . 19
  - 4.3.2. Structure . . . 20
  - 4.3.3. OverlayElement . . . 21
  - 4.3.4. OverlayUiElement . . . 22
  - 4.3.5. OverlayStateManager . . . 23
  - 4.3.6. Panels . . . 24
  - 4.3.7. Popups . . . 26
  - 4.3.8. OverlayWrapperControl . . . 27
- 4.4. Internal Representation of the Mission . . . 28
- 4.5. ROS Nodes . . . 29
  - 4.5.1. Floor Publisher Node . . . 29
  - 4.5.2. Argo Interactive World Model Visualization Node . . . 29
  - 4.5.3. Back-end Planning Node . . . 30
- 4.6. Visual Representation of the Mission . . . 31

5. Evaluation **34**
5.1. Conduction 34
5.2. Results 36

6. Conclusion and Outlook **38**

Bibliography **39**

A. Appendix **40**
List of Figures 40
List of Tables 40
A.1. Survey 41
A.2. Survey Results 44

1 Introduction

1.1 Motivation

One requirement of the third competition in the ARGOS challenge is that a qualified operator without detailed knowledge about the used robot control software can easily create mission plans given in the form of a literal text after a short introduction.
Currently, the mission planning is done directly in FlexBe[1] as seen in 1.1.
While planning in FlexBe is very powerful and flexible, it is also prone to mistakes and not easy to use for someone who has never worked with it before.

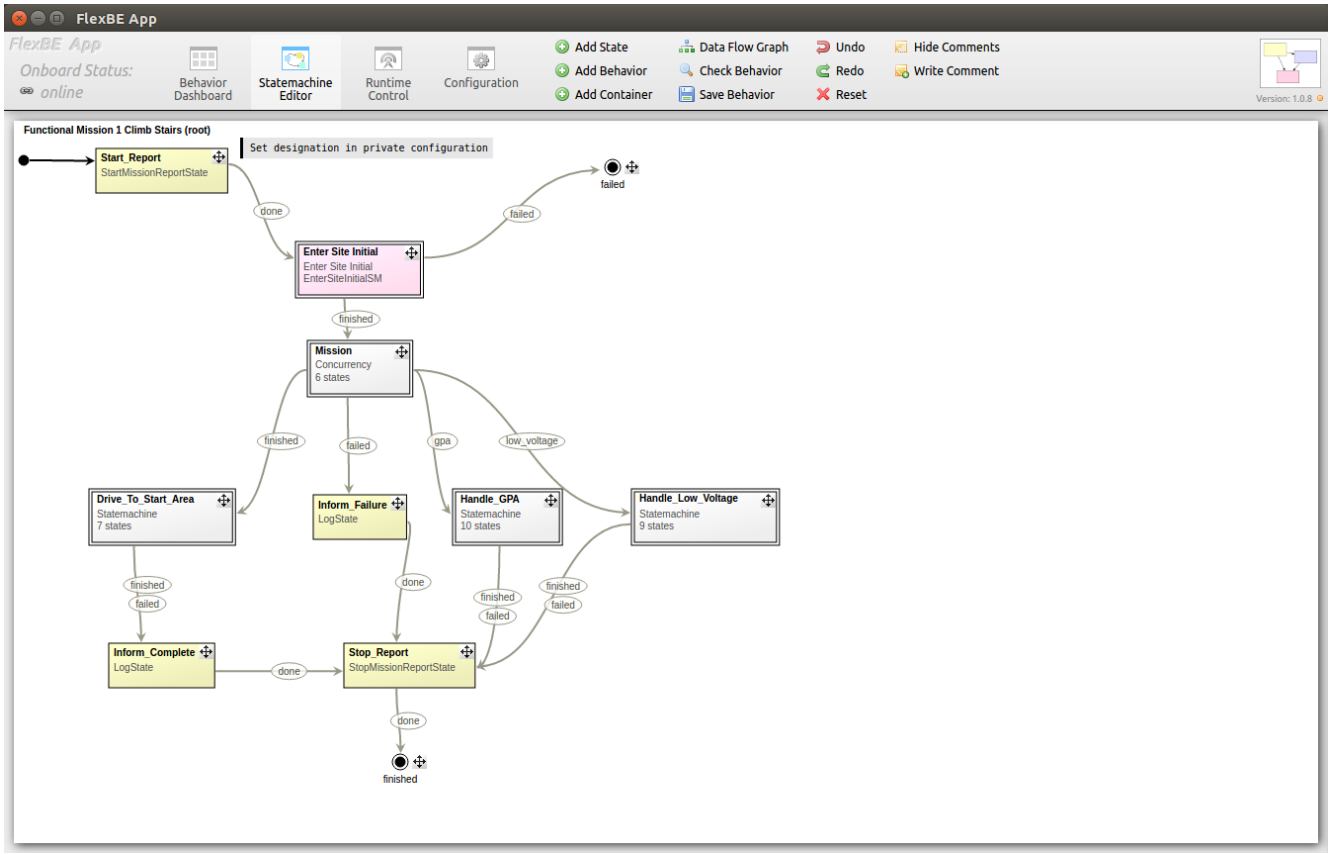


Figure 1.1.: An example of a mission planned in FlexBe.

To maximize the chances of winning the ARGOS challenge, a new planning user interface has to be developed.
An interface that is intuitive and can be introduced to a new user in less than half an hour while still providing enough flexibility to plan even the most complex missions the organizers might think of.

1.2 ARGOS Challenge

The ARGOS challenge, hosted by TOTAL in collaboration with the ANR[1], aims to foster the development of a robot that is able to autonomously oversee the state of on- and offshore oil and gas production sites, essentially, eliminating the need for human workforce in these often very harsh and dangerous environments.

Figure 1.2.: The UMAD site where the competition is held.

In the context of the ARGOS challenge TU Darmstadt paired up with the Austrian company Taurob as Team ARGONAUTS which is one of the 5 teams that were selected by the ANR for funding and participation out of 31 applicants.
Taurob is responsible for the robot's hardware, basic locomotion and teleoperation capabilities whereas TU Darmstadt develops the autonomous robot capabilities and operator-robot interaction[2].

Given that the robots in the ARGOS challenge are developed with the aim to be used by human supervisors on oil production sites, there are several requirements posed on the hard- and software.
The weather on oil production sites, especially offshore oil production sites, can be very harsh. For that reason, the robot has to be able to operate in temperatures ranging from -50°C to 50°C, with wind forces reaching 70 km/h and gusts of up to 100 km/h, operation during day and night with rain, fog, vapor or smoke possibly limiting vision capabilities, a corrosive environment with salty air and a relative humidity of up to 100%.

[1] Agence Nationale de la Recherche - A french institution for the funding of scientific research.

Additionally, it needs to be ATEX[2] certified due to a potentially explosive atmosphere caused by the presence of hydrocarbons.

Considering that the robot is used on a production site which was made for humans and humans might still be on the production site working alongside the robot, there are a few restrictions on the robot.
It needs to be able to climb stairs to get from one floor to another. There is no elevator.
Checkpoints - i.e. pressure gauges, water levels, valves etc. - are reachable for an average human being, which means the robot has to be able to read information from checkpoints located at minimum 0.2 m to a maximum of 2 m with a maximum lateral deviation of 0.5 m relative to the pathway.
Some of these checkpoints might be modified unexpectedly and can vary in their orientation by ±90° and their position may be anywhere in a sphere with a radius of 10 cm around the nominal position.
In the case of an emergency evacuation, the robot should not block the walkway, therefore it should not be wider than 35 cm, a width less than 35 cm actually gives a bonus in the competition, more give a penalty of up to -3% at 70 cm width and if it exceeds 70 cm, it is disqualified.

The "Cart" function - the platform carries a mast or an arm that holds the payload sensors and vital organs.

The "Move" function - the ARGOS robot has to visit checkpoints in a 3D environment, climb and descend stairs, localize itself and find its way, return to the docking station (starting area) or safe areas, etc.

The "Sense & React" function - the payload sensors provide data that has to be processed on the robot online. Depending on the processing result, an appropriate conditional behavior/reaction has to be triggered. Similar associations are expected when a hazard occurs (detection of GPA, obstacle, acoustic gas leak, heat source, and abnormal noise) or other events (low level of batteries, WiFi shutdown or disturbance, emergency stop) or when an inner dysfunction occurs.

The "Interact" function - the function includes the HMI, communications, possible interactions between the robot (in autonomous and in supervisory modes) and humans on the platform. The function plays a major role in supervisory mode, which is the only way in case of emergency situation to control the robot and gather information in a highly degraded environment.

Figure 1.3.: The functional categories, taken from the 'Rules of the 3rd competition'[3]

The robot is required to have two different modes of operation.
In **autonomous mode** the robot autonomously traverses the site, inspects a set of checkpoints consisting of pressure gauges, valves etc. while monitoring for abnormalities like gas leaks, hot spots, different sound signals or alarms and new obstacles which can be positive obstacles, i.e. a stone brick, or negative obstacles, i.e. a missing ground plate.
If the robot detects an anomaly, it is required to report his findings.

[2] The ATEX regulations make sure that equipment is safe to use in a potentially explosive atmosphere. The abbreviation ATEX derives from the french title 'Appareils destinés à être utilisés en **AT**mosphères **EX**plosibles'.

In **supervisory mode**, the operator has high-level control over the robot and can perform actions for which the robot was not programmed or in a degraded environment with the robot still providing assistance functions like collision avoidance and power management.
The operator must always be able to change from one mode to the other at any time and there should be no visible delay, no re-localization, and no information loss.

In three challenges the robot systems are evaluated using several criteria based on reliability, safety, and robustness.
The robots main functions are separated into four categories as described in figure 1.3.

Mission's designation:	Mission #1
Main mode:	Autonomous
Maximum mission total duration:	20'
MISSION PREPARATION PARAMETERS:	
Anomaly specification:	Default normal values/positions/levels for the checkpoints (as specified in the rules, appendix A) **Real-time reporting:** As per the rules **After-mission report:** As per the rules
Initial position:	The robot is docked and connected in its docking station in the "Starting area" 1 or 2, specified by the ARGOS staff during the mission preparation.
End of the mission:	The mission ends when the robot is docked and connected in the docking station in the "Starting area" 1 or 2, specified by the ARGOS staff during the mission preparation.
The robot will have to control checkpoints in this precise order: 1. Checkpoint #1: read the value on pressure gauge 1 2. Checkpoint #4: read the position of valve R25 3. Checkpoint # 8: read the value on pressure gauge 5	

Figure 1.4.: The script of the 1st mission of the 3rd competition.

A challenge usually consists of multiple missions. An example of the script for a mission is given in 1.4.
In this mission, the robot starts at one of the two starting areas as defined in the competition rules, has to enter the site and control the checkpoints 1, 4 and 8 autonomously.
Afterward, it has to leave the site again and drive back to the docking station in the start area.
During these missions, there can be anomalies as mentioned above and the robot has to detect them and act appropriately.

The first competition focused mainly on the first two functions (CART and MOVE), the second competition tested the improvements of the first two functions and some aspects of the functions SENSE & REACT and INTERACT, and the last competition will test all four functions in detail in scenarios derived from realistic situations when the robots are deployed in on- and offshore production sites.

Apart from these main functions, there are lots of requirements for how the robot should react to certain conditions, which also differs depending on the robots operational mode.

As for the results of the challenges, the first challenge is not taken into account, the second competition accounts for 25% of the final score, while the third – and last – challenge makes up for the remaining 75%.
As of now, the first two challenges already have taken place and in the official ranking, Team ARGONAUTS is ranked second with the French Team VIKINGS being in the lead.

1.3 Robot Operating System

The *Robot Operating System*[4], or short *ROS*, is a framework providing all the tools and libraries needed to create complex and robust robot interaction and behavior. It was designed to simplify the creation of modular robot software components that can be written in any programming language that supports networking.

Components in *ROS* are called *Nodes*.
A *Node* is essentially a standalone piece of software that can communicate with other nodes to achieve its goal using *ROS* APIs[3].
ROS also provides a parameter system in the form of a global key-value store, meaning nodes can have parameters that influence their behavior and make them even more flexible.

ROS supports three different kinds of communication:

- Message Passing
- Remote Procedure Calls
- Preemptable Remote Procedure Calls

Message Passing is done using a *Publisher* and any number of *Subscribers*.
A *Publisher* is created by advertising a certain type on a topic. A topic is a named bus over which messages can be exchanged.
Topics are strongly typed and *Subscribers* will only establish a connection if the types match.
However, they do allow for multiple *Subscribers* and even multiple *Publishers* on the same topic.
Optionally connect and disconnect callbacks can be provided, a queue size that determines how many outgoing message can be queued for delivery until some have to be dropped and whether or not the *Publisher* is latched can be set. A latched *Publisher* always sends the latest message to new subscribers.
Subscribers listen for these messages and whenever the *Publisher* publishes a message, a callback is executed on each *Subscriber*.
Aside from the obvious advantage of a well-tested and easy-to-use messaging system, it also forces developers to define clear interfaces between the single components, resulting in better and more reusable code.
The content of these messages is declared using an *Interface Description Language*, the message IDL.
ROS manages all the details of communication such as the distribution to all subscribers, the serialization and the deserialization of messages.

Remote Procedure Calls are called *Services* in *ROS*.
A node can provide any number of services by creating an instance of *ros::ServiceServer* for each service. These services are also advertised on a topic and have a callback that is executed on each service call.
A service callback has to return a boolean indicating whether or not the service call was successful. It is passed two parameters by reference, a request and a response that is sent back to the caller after execution.

[3] Application Programming Interface

Services are just like messages declared using the message IDL.

Preemptable Remote Procedure Calls are called *Actions* in *ROS*.
Sometimes an operation can take some time, i.e. driving to a checkpoint. For such an operation it would be good to monitor the progress of the operation or even cancel it along the way.
For this purpose *ROS* provides *Actions*.
Actions are essentially like *Services* with the exception that they can report progress during execution and be canceled by the caller.

1.4 RViz

RViz[5] is actually an *ROS* tool.
In fact, it is even one of the most well-known tools in *ROS* according to *ros.org*.
However, considering its important role for the user interface presented in this thesis, it deserves its own section.

RViz is a general purpose three-dimensional visualization tool for *ROS*.
It has a very rich set of built-in plugins to visualize common message types in *ROS* including laser scans, point clouds, and camera images.
This renders *RViz* an incredibly useful tool for debugging. Being able to see what the robot sees is of great help when trying to figure out why the robot behaves the way it does.

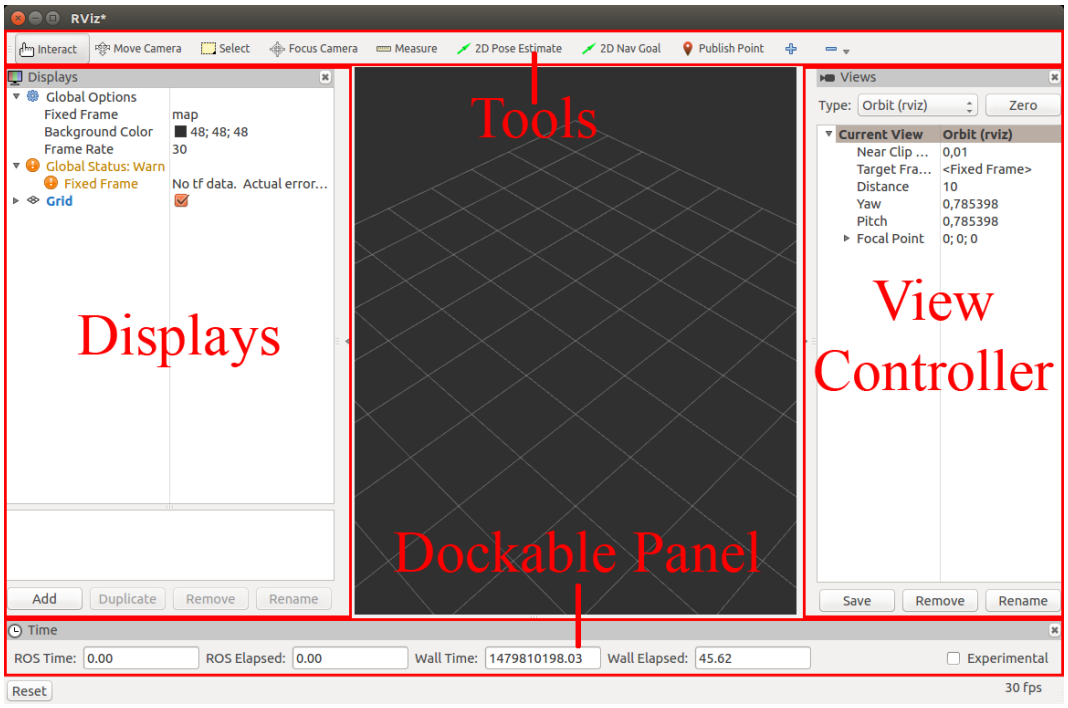


Figure 1.5.: The different kinds of plugins in *RViz*

In this thesis, *RViz* is going to be used as a base and extended using its extensive plugin system.
RViz supports four different kinds of plugins as shown in fig 1.5:

- Tools
- Displays
- View Controllers
- Dockable Panels

All these different kinds of plugins except for *Dockable Panels* can use *RViz* properties to allow the configuration of the plugin – see figure 1.5 (Displays & View Controller).
Properties are automatically saved to and loaded from the *RViz* config file.

Tools usually have a single purpose.
For example, there is the default *Interact* tool which enables the user to interact with interactive markers etc. or the *Select* tool which can be used to select objects in the 3D-view.
A tool can be selected using the tool menu at the top or if supported by the tool, using a hot-key.
Basically, which tool is used determines how the user interacts with the 3D-scene.
The default tools mainly affect the mouse interaction, though, they can also react to keyboard events and even add elements to the scene.

Displays essentially add support for the visualization of *ROS* messages to the 3D-scene.
For example, the default *MarkerDisplay* can be set to a topic that publishes *Marker* messages, and display these markers in the 3D-scene

View Controllers determine how the 3D-scene is viewed and the interaction with the camera.
For example, the default *ViewControllers* include an *OrbitViewController* where the camera orbits a fixed point in the scene and when moving always stays focused at the fixed point.
The *FPSViewController* is quite the opposite where the camera position is fixed and similar to **F**irst-**P**erson **S**hooter games you can only move the camera with regard to its orientation.

Dockable Panels are basically Qt[4] widgets that can be docked anywhere in the *RViz* window.

1.5 Ogre

OGRE3D[5] is a scene-oriented, open-source 3D graphics rendering engine used by *RViz* to visualize the 3D-scene.
In the context of this thesis, only the *OverlayManager* is important as it can be accessed as a *Singleton*[6] class which can be used to render a material with a given texture on top of the 3D-scene.
This essentially means that using the *OverlayManager* one can draw an image on top of the 3D-scene.

[4] See section 1.6
[5] *OGRE - Open Source 3D Graphics Engine*. URL: `http://www.ogre3d.org/` (visited on 11/13/2016).
[6] A singleton class is a class that is restricted to one instantiated object that is usually accessed globally.

1.6 Qt

Qt is a widely used cross-platform application framework.
It powers a wide range of applications both commercial and non-commercial including the aforementioned *RViz*.
Qt also adds a lot of useful extensions to the C++ language.
One of the most important features and also heavily used in this thesis is the communication between objects using *signals* and *slots*.
Classes using *signals* or *slots* have to extend the *QObject* class and need to be preprocessed by the *Qt Meta-Object Compiler* which generates the C++ code necessary to provide the functionality of *signals* and *slots*.

A **signal** can be sent by *Qt* objects and can contain event information.
A **slot** is a special function that receives the emitted *signal* and the provided event information.

2 State of Research

To the best of the author's knowledge, there are no publications on user interfaces or even mission planning for single autonomous inspection robots with a focus on an intuitive user experience available.
Most papers focus on multi-agent systems or they only plan one fixed mission in code.

There is of course other software for mission planning of single autonomous devices.
For example, *Mimosa2* - a tool for the preparation, supervision and analysis of IFREMER's subsea vehicle - and QGroundControl - a tool for flight control and mission planning for drones.
Though there is no paper on their design or implementation available and the type of device they were developed for is inherently different from the robot that is used in the context of the ARGOS challenge.

Of course, there is even software under active development for the exact same purpose.
However, this software is developed by Team ARGOS competitors in the challenge and not publicly available.
The only information that can be found publicly on the internet about the mission planning of the other teams is the image of Team LIO's mission planning in figure 2.1, which is taken from their presentation at ROSCon 2016[7].
They were also using a hierarchical state automaton similar to our back-end FlexBe, anyhow, it is safe to assume that they are working on a more intuitive user interface for the mission planning as well.

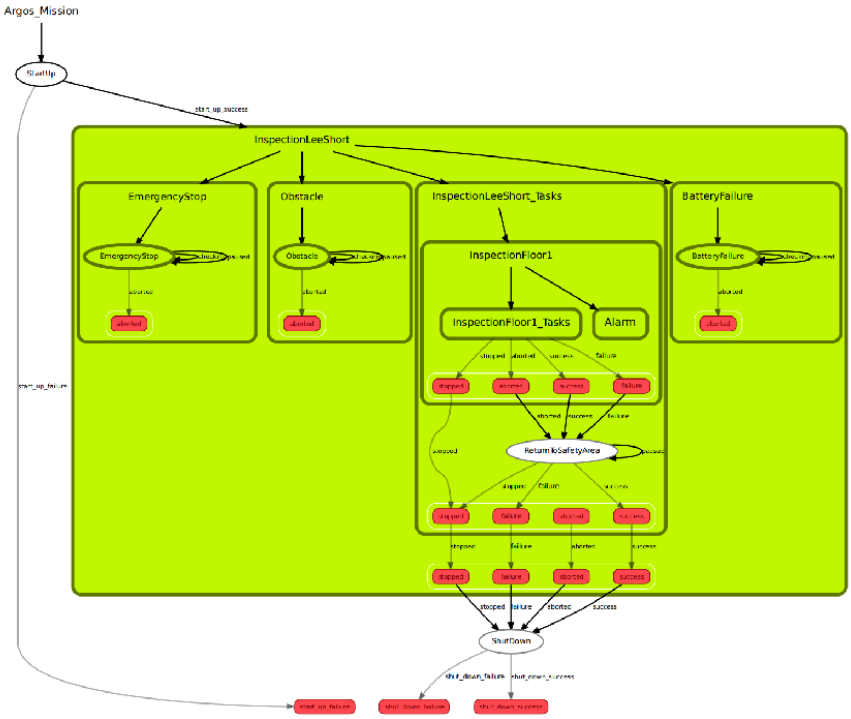


Figure 2.1.: Mission Creation for Team LIO's ANYmal.

3 Concepts

3.1 Visual Representation of the Mission

One of the greatest challenges of developing a planning user interface is the presentation of the planned mission to the user.
It has to be represented in a way that the user can intuitively understand whether the planned mission is equal to the mission he envisioned or not.

Fortunately, there is another type of software that has had the need to represent planned actions intuitively to a broad audience for over 15 years.
Video games – or to be more specific strategic video games – feature a sometimes more, sometimes less sophisticated implementation of mission planning.
In basically every good strategy game, the ahead planning of tasks is a very important component and years of development went into improving it, thus, they are a great place to look for ideas.

However, most of them are still real-time which means that the player does not *plan and execute* but rather *plans* ***while*** *executing*.
This is a key difference to the purpose of the user interface developed within this thesis.

Figure 3.1.: Mission planning in strategic video games (Planetary Annihilation).

When displaying the planned mission, it should be as close to the actual execution of the mission as possible.
This includes a visualization of the path the robot is going to take to get from task A to task B.

Additionally, planned tasks have to be clearly distinguishable from the current position.
There are different approaches to accomplish this.
For example, the planned task may be colored differently and/or it is displayed semi-transparent

by applying an alpha value.
Another method is to reduce the visual details and only display a skeleton which is done in figure 3.1.
For the envisioned user interface, an alpha approach is chosen to distinguish planned tasks from the task that is currently selected.

Considering that a mission can get quite complex, it should also support showing only parts of the mission – i.e. by limiting the number of displayed tasks to the last 5 tasks before and after the current and/or grouping the tasks and provide the possibility to toggle the entire group's visibility.

3.2 Internal Representation of the Mission

Another issue with the mission planning in FlexBe is that it is normally meant to be used as the robot's only controller.
However, in the context of the ARGOS challenge, we need to be able to switch to supervision – maybe even do a part of the mission under supervision – and when switching back to autonomous the mission should continue with the remaining tasks.

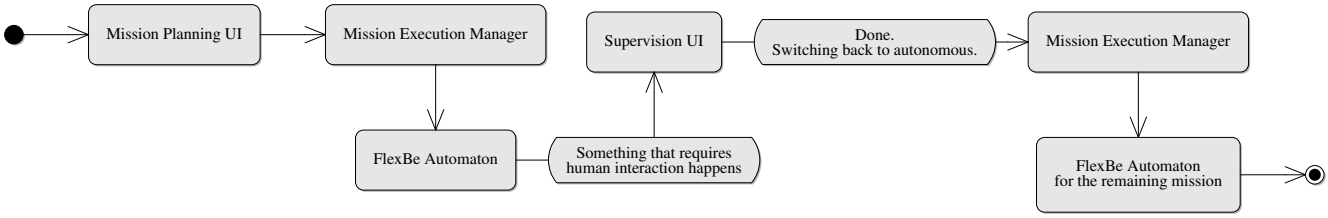


Figure 3.2.: An exemplary mission flow.

FlexBe does not support such a behavior and though there might be ways to overcome this and to build the FlexBe automaton in such a way that tasks that were done under supervision can be skipped, this would be a very ugly and fragile solution.

To solve this problem and to provide a more robust solution, an intermediate language (IL) had to be developed.
By using an IL as a layer of abstraction, the mission state can be tracked during both autonomous execution and supervision.
Now, upon starting the mission a FlexBe automaton is generated from the IL. This is done by a separate node, in the following paragraphs referred to as synthesizer.
If user interaction is required and therefore a switch to supervision is necessary, the FlexBe automaton is simply stopped, what has to be done is done in supervision and afterward, when switching back to autonomous, a new FlexBe automaton consisting only of the mission's remaining tasks is generated.

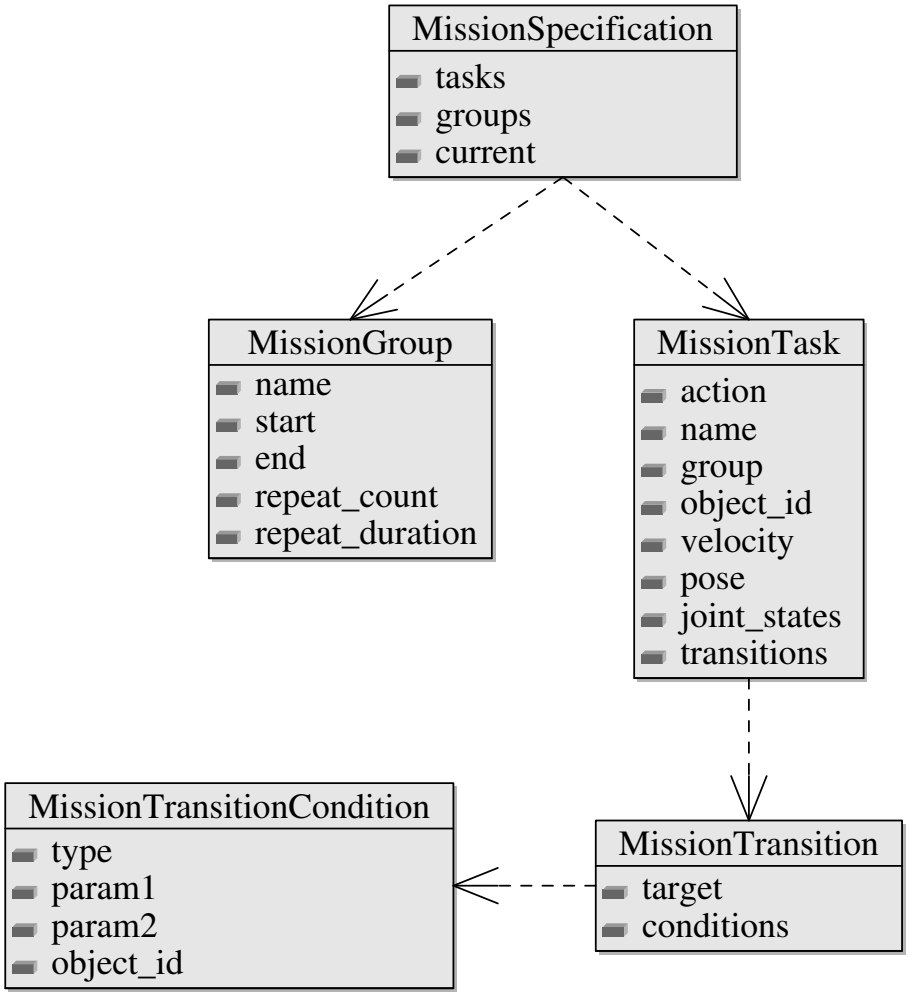


Figure 3.3.: The message structure for the intermediate language.

For the IL, the planning of missions is abstracted to the structures displayed in 3.3.
A mission consists of a set of tasks and optionally any number of groups.
The attribute current is only relevant for the mission planning user interface. It is used to visually set the current task apart from the other planned tasks. Also, new tasks are always inserted after the current task.

Groups can, for example, be used to create an inspection round and repeat it for either a fixed amount of repetitions, a fixed duration or both.
They were intentionally separated from the tasks to make it more robust.
The other option would have been to add a transition from the group's end back to the start task. However, the only advantage this would offer is that synthesizing might be a bit easier because the synthesizer would not have to explicitly check if the task is the end of a group.
On the other hand, the synthesizer would have to handle the groups whereas, in the chosen approach, a simpler synthesizer can choose to ignore groups.
Another advantage is that the synthesizer can decide when to leave the group – i.e. whether it wants to hard stop the group and proceed with the mission as soon as the time limit is hit or soft stop and complete the group one last time.

Tasks consist of an action which is a predefined constant.
Currently, the available actions are limited to the following:

- START
- WAYPOINT
- MEASURE

Most of these should be self-explaining except for *START* which is only required for the planning user interface to display the starting point and should be ignored by the synthesizer.
Apart from the action, a task also has a name which has to be unique because this is how the transitions – which are covered in the next paragraph – identify their target and how groups can specify their start and end task.
It also has a group which is mainly intended for displaying purposes, an optional object_id which is required by the *MEASURE* action, a velocity which specifies how fast the robot should drive to the next task's position, an optional pose required by the *WAYPOINT* action and of course a list of transitions which specify how to continue the mission.

Transitions are pretty straight forward. They only have a target which is – as already mentioned – the task id the transition points to and a list of conditions which have to be fulfilled in order for the transition to trigger.
It is important to note that all of the conditions have to be met and it is also possible to leave the conditions list empty. If the condition list is empty, the transition is unconditional.
A task can have only one unconditional transition and it also should always have one unless it is the mission's end.
Regardless of where the unconditional transition is located in the list, it is only executed if no other transition can be made.

Conditions consist of a type, an id, an object_id and two optional parameters, namely param1 and param2.
As of now, the type can have one of the following values:

Type	Required Parameters	Expression
MEASURED_VALUE_LESS	object_id, param1	**val**(object_id) < param1
MEASURED_VALUE_GREATER	object_id, param1	**val**(object_id) < param1
MEASURED_VALUE_WITHIN	object_id, param1, param2	param1 < **val**(object_id) < param2

Table 3.1.: The different condition types.

3.3 User Interface

Another important aspect is the displaying of information and controls for user interaction.
The displayed elements should be limited to the most necessary and grouped by their type in order to keep the user interface as clean and intuitive as possible.
Elements that are not used frequently during normal usage should be hidden through either an expandable section or a popup opened by a button.

To maximize the use of available space, the information, and basic controls should be rendered as an overlay on top of the 3D-view of the planned mission.
Besides, this also gives the user interface a nice modern and sleek look.

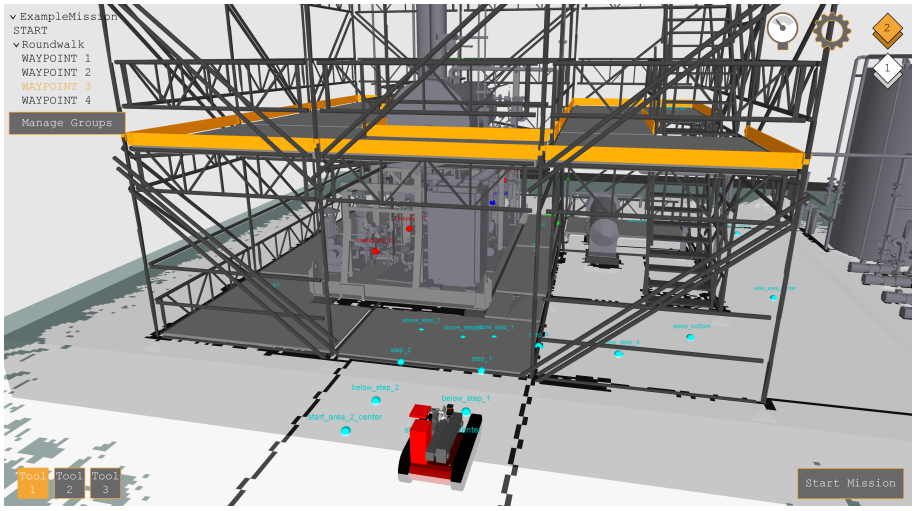


Figure 3.4.: A concept of the user interface.

Figure 3.4 shows the concept of this overlay.
In the top left, the planned mission is displayed.
It consists of a *START* followed by a group called 'Roundwalk' consisting of four waypoints.
The currently selected task *WAYPOINT 3* is highlighted using the accent color – see 3.4.
Single groups like *Roundwalk* or the mission as a whole can be collapsed if not needed.
The area used to display the mission can grow up to a third of the vertical space. If it exceeds that limit, it becomes scrollable.

In the bottom left, the available tools are displayed. *Tool 1* is the currently active tool which is highlighted by using the accent color as the background.

In the top right, we have the floor selection with the second floor being highlighted as the currently selected option. To the left, we have two buttons. The left one opens a menu with all checkpoints for easier navigation and the right one opens a settings menu where i.e. the buttons to create, load and save a mission can be found.
Checkpoints are hidden in a menu even though they are an important part of mission planning because there are just too many of them to display at all times.

Finally, a button to start the currently planned mission is located in the bottom right corner. If the node responsible for executing the planned mission is not available, the button should change to a 'Save Mission' button and only store the mission for later use.

3.4 Design

A small glimpse of the design was already shown in figure 3.4.
In this section, the underlying design considerations and choices are detailed.
The three key aspects kept in mind for each and every design choice were **Usability**, **Intuitiveness**, and **Responsiveness**.

Another key aspect of choosing the right design was Karim Barth's Bachelor Thesis 'Development of a User Interface for Supervised Inspection using mobile Robots in challenging Environments' [8] considering his supervision user interface and the planning user interface introduced in this thesis are part of one user experience: The planning and execution of missions in the context of the ARGOS challenge.
When starting the mission the GUI (Graphical User Interface) switches to the supervision user interface for the execution of the mission and because changing the planned mission during execution is a requirement, the supervision user interface also features a button to switch back to the planning user interface.

This, of course, means that for a good user experience our interfaces should have a consistent look.
In the optimal case, the user will not even notice that they are two separate user interfaces.
To achieve such a unified look, definitions for the most common design resources – i.e. colors, fonts, margins and paddings etc. – were created.

A sense of **Responsiveness** is created by using different background colors and/or in some cases foreground colors on all controls that support user interaction for each state – with the states being: *default*, *mouse-over*, *mouse-pressed* and for controls that can be toggled additionally a *checked* state.

Good **Usability** is created by limiting the at-all-times-visible controls to only the most important. Not frequently used controls are only displayed when needed and can be reached through as few button clicks as possible.

> „A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."
>
> – Antoine de Saint-Exupéry. *Terre des Hommes*. 1939

Intuitiveness, on the other hand, is not easy to quantify.
In an effort to make the interface as intuitive as possible, the used icons and button texts should be very expressive.
However, as a developer, one can only create an interface in a way oneself finds intuitive, which is not necessarily what someone else would find intuitive.
To make sure other people feel the same, feedback and testing by as many individuals as possible is required.

4 Software

4.1 Implementation

In the following sections, details about the implementation of the planning user interface are provided.
It uses *RViz* as a base for its visualization and is implemented as a set of *RViz* plugins, *ROS* nodes, and *ROS* services.

4.2 General Displays, Tools and View Controllers

4.2.1 Mouse View Controller

The **Mouse View Controller** is a *View Controller* plugin, which – as mentioned in section 1.4 – controls how the user interacts with the camera.
It is based on the *Animated View Controller* – developed by Adam Leeper[10] – which provides methods to move the camera slowly to a given location instead of jumping there instantly.
Considering that *RViz* is actually meant to be a visualization tool, it comes as no surprise that the default *View Controller* plugins only allow to either look at a fixed focus point that is orbited by the camera or a fixed camera where only the orientation can be changed.
RViz does allow to move within the 3D-scene, but it's far from intuitive and simple.

However, for a planning user interface the camera movement within the 3D-scene is an important part.
In order to make it more intuitive, the **MouseViewController** was created.
It allows using the mouse to control the camera by moving near the edges and has two modes that can be enabled or disabled using keyboard shortcuts.

Mode	Description	Shortcut
Auto-Lock	The mouse is locked within the *RenderPanel*[1] but can escape if the velocity is high enough. It will automatically be locked again upon reentrance.	Ctrl + A
Locked	The mouse is locked within the *RenderPanel* and can not escape it. If it tries to leave, it is moved back to the closest point within the *RenderPanel*.	Ctrl + L

Table 4.1.: The different lock modes supported by the **Mouse View Controller**.

In both modes whenever the mouse comes close to the border of the *RenderPanel*, the camera moves in the corresponding direction – which depends on the camera orientation.

[1] The *RenderPanel* is the area in which the 3D-scene is drawn.

The width of the area in which the mouse triggers camera movement as well as the maximum camera velocity can both be adjusted using *RViz Properties*.
Though, the name **Mouse View Controller** is a bit misleading because it does not solely accept mouse input but also supports camera movement using the arrow keys.
The movement speed can be doubled by holding the shift key while moving.

This functionality is implemented by installing an application-wide *Qt* event-filter.
It evaluates the arrow keys and keeps the mouse within the *RenderPanel*.
However, the movement itself is done separately in the plugin's *update* method that is called every time the 3D-view is updated and passed the elapsed time since the last update, which allows for a smooth camera movement.

4.2.2 Tool Selection Display

The **Tool Selection Display** is a *Display* plugin which extends the *OverlayDisplay* that will be introduced in section 4.3.
As shown in figure 3.4, it displays the available tools as radio buttons in the bottom left of the screen and highlights the currently selected tool.
It does this by querying the available tools upon initialization using the *rviz::ToolManager* and connecting to the *Qt* signals that are emitted whenever a tool is selected, added or removed.

This display was added for three reasons:
Firstly, the operator does not have to leave the *RenderPanel* in order to select a tool, which greatly improves the usefulness of the **Mouse View Controller**.
Secondly, we use the available display space more efficiently and get approximately 30px more vertical space or roughly 3% on a 1080p monitor.
And **lastly**, it blends in better with the overall design.

4.2.3 UMAD Floor Control

The **UMAD Floor Control** is also a *Display* plugin extending the *OverlayDisplay*.
It shows floor icons stacked in the top right corner of the *RenderPanel* and communicates with a background node that is responsible for publishing only the currently selected floor and all floors below it using service calls.

4.2.4 Robot Navigation Tool

The **Robot Navigation Tool** is a *Tool* plugin.
It allows placing a robot in the 3D-scene with a given orientation that can be set by using the scroll wheel.
The target position is obtained using a method of the *SelectionManager* which takes a 2D-point – in this case, the position of the mouse cursor – and returns the corresponding 3D-point of the

object that is rendered topmost at the given position.

When the tool is active, it displays a preview of the robot at the given position and orientation, so that the user can see beforehand if the waypoint can be reached or collides with the site. On mouse click, the pose consisting of the position and orientation is published to the topic:

/argo_robot_navigation_tool

This allows it to be used to set waypoints for both the planning and supervision.

4.2.5 Place Obstacle Tool

The **Place Obstacle Tool** is a *Tool* as well.
It allows adding an obstacle to the UMAD site manually.
As in 4.2.4, the 3D-point corresponding to the 2D cursor position is obtained using the *SelectionManager*.
On the first mouse click, the start of the obstacle is set, the second click sets the end and places a roadblock from start to end.
The placed obstacle is added to the world model using a service which currently only supports obstacles as single points without information on their size or shape.
However, a new obstacle server is planned and once it is implemented, the tool will be updated accordingly.

4.3 Overlay

4.3.1 Rendering the Overlay

As mentioned in section 3.3, all information that can be displayed as an overlay should be displayed as an overlay.
Unfortunately, *RViz* does not provide a simple method to display information at fixed positions in the 3D-view.
To overcome this a solution had to be found.

Various methods to achieve the goal of overlaying information on top of the 3D-scene were investigated. All were based on the same method of drawing on top of the 3D-scene using the aforementioned *OverlayManager* provided by *OGRE3D*.
To initialize the overlay, first an *Ogre::Overlay** and *Ogre::OverlayContainer** are created using the *OverlayManager*.
The container is set to span across the entire scene.
Then an *Ogre::MaterialPtr* is created and linked to the container.
Lastly, the container is added to the overlay.

Now, whenever the overlay is drawn, the size of the *RenderPanel* responsible for rendering the 3D-scene is obtained and if it changed, a new texture with the given width and height is created and applied to the *Ogre::MaterialPtr*.
The texture buffer is obtained, locked and cleared. Then, a *QImage* is created on the now empty texture data.
Finally, the controls are rendered inside the *QImage* and the buffer is unlocked.

The first attempt to simply render *Qt* inside the *QImage* worked pretty well after some tweaking. This would have been the preferred approach because it would allow using an existing and very powerful application framework.
However, the mouse and keyboard interaction did not work at all.
After a week of trying different methods to pass the mouse and keyboard interaction to the overlay controls, the approach described in the following was used instead.

The actual implementation is now a lightweight library that was developed as part of this thesis.
It uses some components of *Qt* and the *Boost* libraries because they are used by *RViz* and therefore available without additional overhead.

4.3.2 Structure

As shown in figure 4.1, all elements are sub-classes of the *OverlayElement* class.
It provides some basic properties necessary for all kinds of elements.
Panels and controls are additionally sub-classes of the *OverlayUiElement* class.
The reason for this is explained in 4.3.4.

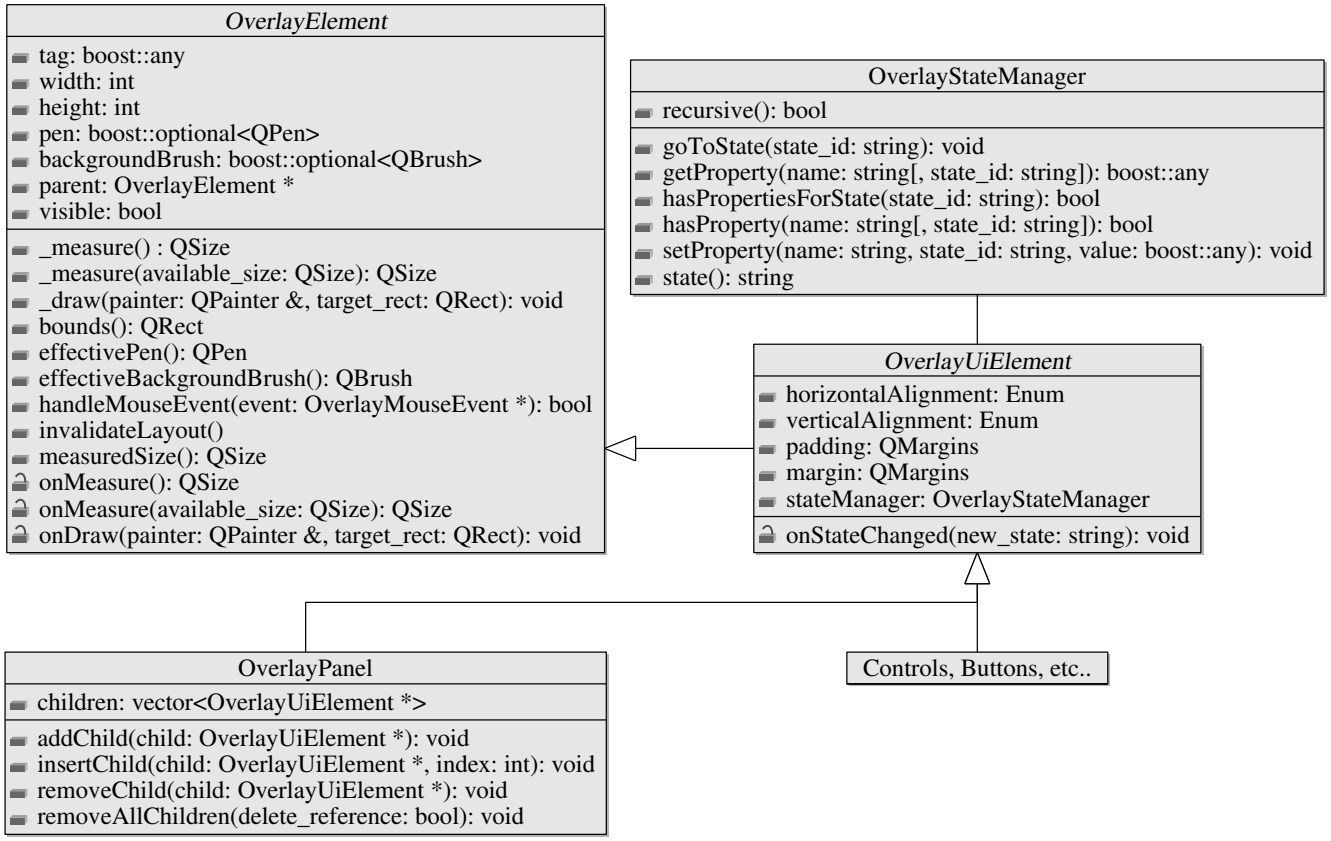


Figure 4.1.: Part of the structure of the overlay's elements.

4.3.3 OverlayElement

The **width** and **height** are pretty straight forward and represent the desired width and height of the element, which can also be set to the constant *Auto* with the value -1. If the width or height is set to *Auto*, it is determined by the content of the element.
The **tag** is just a general purpose field that can be used to associate any kind of information with the control.
Pen and **background** are the foreground and background of the element.
As the alert reader might have noticed, these properties use the type *boost::optional* to store their values. This is because they are meant to propagate down the element tree.
When drawing, the element always calls the *effectivePen()* and *effectiveBackgroundBrush()* methods.
As depicted in figure 4.2, these methods check if the *boost::optional* has a value and if not, they check if it has a parent. If the element has a parent, it is asked for its effective value, otherwise, a default value is returned.

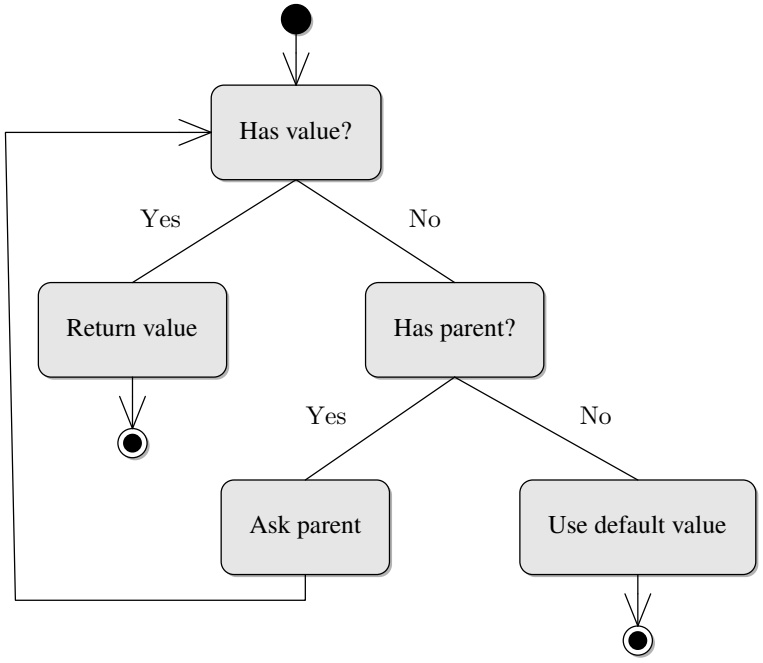


Figure 4.2.: How the effective value of the pen and background property is determined.

The *OverlayElement* also defines a few functions used to render and interact with the element.
The **handleMouseEvent(...)** method takes a mouse event and returns a boolean indicating whether it consumed it or not. If it consumed the event, all future events will first be sent to the consuming element until it returns false or the event got canceled - i.e. because an element with a higher priority took the event.
For the sake of readability, the mouse functions in the figure were reduced to only this most important and generic one. However, there are of course a few more handlers including the most common types of interaction: *mouse over* and *mouse pressed*.

For the rendering, the element has to first determine its size before actually drawing it.
For this, there are two **_measure** functions. One that has no parameters and returns the minimum desired size of the element and another one that is given the available size and returns the size it wants.
These methods are not *virtual* but call the *protected virtual* **onMeasure** methods.
This is because they also handle caching the result for better performance.
It is assumed that the measured size will not change if the available size stayed the same and the layout was not invalidated by calling the **invalidateLayout** method, which invalidates the layout of the element and propagates to all parent elements.
Also, they start with a '_' to mark them as internal methods that should only be called if the developer knows exactly what he is doing.
Though, unless one is developing a new primitive control[1], calling this method is almost always unnecessary and may result in unexpected results.
It should also be noted that the *_measure(QSize)* method allows the passing of the constant *INT_MAX* as width or height, to indicate that the element may take as much space as it needs. However, the *_onMeasure(QSize)* method is not allowed to return *INT_MAX*.

Finally, the element is drawn by calling the **_draw** method, which internally calls the *protected virtual* **onDraw** method and makes sure the element is not drawn if it is not visible and saves the provided *QRect* as the bounds of the element. The bounds are required internally for the interaction to determine if the mouse is over an element.

4.3.4 OverlayUiElement

As seen in figure 4.1, all overlay controls inherit from another class *OverlayUiElement* that inherits from *OverlayElement*.
The reason for this additional level in the hierarchy is that there may be elements that are rendered in the overlay but not part of the main element tree.
For example, the *Popups* covered in section 4.3.7. They are meant to be rendered on top of the overlay and are not part of the element tree.
To ensure that these elements can not be added to the element tree, the class *OverlayUiElement* was added, which also adds some properties and methods that are only required for elements that are part of the element tree.

The **horizontalAlignment** determines whether the element is placed on the left, right or stretches to the available size if there is more horizontal space than it needs.
verticalAlignment is the equivalent of **horizontalAlignment** for the vertical space.
The **padding** and **margin** properties determine how much free space should be reserved on the in- and outside of the element's border.
As depicted in figure 4.3, the *padding* is part of the element and has to be included in the measurements done in the *onMeasure* methods of the element itself. The *margin*, on the other hand, is not part of the element and is handled by the parent element.
Lastly, the **stateManager** will be covered in section 4.3.5.

[1] See section 4.3.8

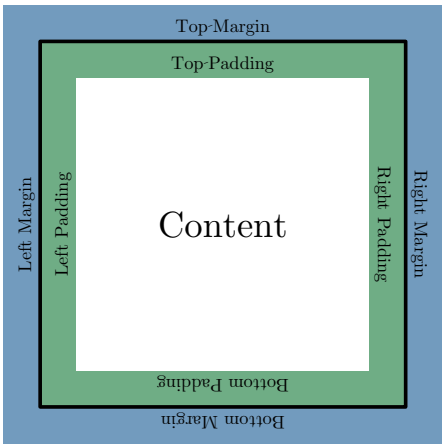


Figure 4.3.: The effect of margin and padding on the layout.
The black rectangle marks the border of the element.

All basic elements like *OverlayText*, *OverlayButton* etc. inherit from this base class as well as the *OverlayPanel* which is a base class for the different layouts introduced in section 4.3.6.
The controls will not be covered in this thesis as they are essentially behaving in the same way as they do in other application frameworks.
There are primitive controls to display texts, rectangles, images, buttons, sliders, radios etc.
The event system is based on *Qt signals* and *slots* – i.e. to connect to the *clicked* event of a button, a *slot* matching the signature of the *signal* has to be created in the receiving class and connected to the *clicked signal* of the button.

4.3.5 OverlayStateManager

The **OverlayStateManager** is responsible for managing the state of the element.
States are represented as strings, which makes them easily extensible.
For each state, a key-value-pair consisting of the property key and a value for the property can be set.
If the element supports the key and is currently in the given state, it will replace its current value of the given property by the value that is stored in the *OverlayStateManager*.
This allows for a very flexible change of the element's properties – i.e. changing the background brush to a prominent color if the element is in a *selected* state.

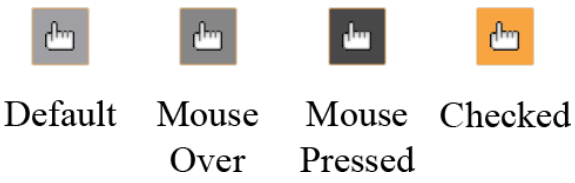


Figure 4.4.: An example of how the different states can be used to visually represent an element's state.

The predefined states include *default*, *mouse-over*, *mouse-pressed*, *checked*, *checked-mouse-over* and *checked-mouse-pressed*.

State changes are propagated downwards on most controls. For example, if a button changes to its *mouse-over* state, the child element and its potential element tree also change to the *mouse-over* state.

4.3.6 Panels

Basically, the overlay's heart as they control how the elements are arranged and also manage most of the mouse interaction.

OverlayPanel

The most basic panel, which is also shown in figure 4.1, and the base for all other panels, is the **OverlayPanel**.
Unlike the *OverlayElement* and the *OverlayUiElement*, it is not an abstract class but actually a fully functioning panel.
It provides methods to manage its children and basic mouse interaction.
The minimum size is simply the minimum size of its biggest element.
Elements can be aligned and have a margin to keep distance to the panel's borders.
Though, they do not interact with each other.

OverlayStackPanel

One of the most important panels for UI design as it allows to simply stack elements horizontally or vertically.
When measuring, the *OverlayStackPanel* passes its child depending on its orientation either a width (Horizontal) or a height (Vertical) of *INT_MAX*.
To keep a consistent amount of free space between each element a spacing can be set.

OverlayGridLayout

The most complicated but also most powerful panel of the three panels provided by this overlay library.
It allows creating a grid layout with a set of row and column definitions. Row and column definitions are structs with a double indicating the height / width, a unit, a minimum and a maximum height / width.
The different units are explained in table 4.2.

For each child, the row and column but also a row-span and a column-span can be specified.
The layout is calculated by first setting all calculated widths and heights to zero.
In the next step, first, the *Pixel* rows and columns are calculated since their heights and widths do not depend on any other factor.

Mode	Description
Auto	The row / column will grow to fit its content. The value of the double is ignored.
Pixel	The row / column will have the value of the double as height / width.
Weight	The value of the double is the weight, the row / column has when the remaining space is distributed. For example, if there are 300px left after processing the Auto and Pixel rows and row 1 has a weight of 2.0 and row 3 a weight of 1.0, row 1 will be 200px high and row 3 will be 100px high.

Table 4.2.: The different units of the row and column definitions.

After that, in a loop the *Auto* cells are updated and the remaining space is distributed between the *Weight* cells.

The loop repeats as long as the layout changes with a maximum of 10 times before printing a warning and stepping out of the loop.
This is because depending on how much space in width or height a cell can actually get its desired size might change.
For example, a text can be wrapped, but when first measuring the width and height of the final cell is not known and it will return the width and height of the text if written in one line.

Now, after we have determined the first layout, we go back to the text element and this time we know that it will not get enough horizontal space to write the text in one line, which means it has to increase its desired height.

The maximum attempt count of 10 was deemed a good trade-off between performance and getting the desired layout. It generally should not happen that the layout changes 10 times.

Once the layout is calculated, every element is measured again with the final size of the rows and columns it occupies.

OverlayItemsGrid

The previously introduced panels already cover a broad range of usage scenarios.
What's still missing is a panel that can display a variable amount of elements not wide enough to be displayed in an *OverlayStackPanel*.
For this reason, the **OverlayItemsGrid** was created.
It stacks elements horizontally or vertically depending on its orientation property and whenever a row or column is full, a new one is added.
The size of the panel's children is determined by measuring their minimum size and passing it to their *_measure(QSize)* method.

4.3.7 Popups

They are not part of the element tree, which is why they only inherit from *OverlayElement* but not *OverlayUiElement*.
Popups are displayed on top of the other overlay elements and have a higher priority with regard to mouse events.
Basically, popups are just a container for a separate element tree. They all have a common base class, the *OverlayPopupBase* which handles the basic functionality like measuring and positioning the content, as well as to provide basic mouse handling.

The base functionality includes a boolean property indicating whether or not the popup is light-dismissable – which, if true, means that the popup closes if the mouse clicks anywhere outside of its bounds.
Mouse events that are within the popup's boundaries are passed to the popup's content and handled accordingly as described in section 4.3.3.
However, if the mouse event was within the popups bounds, the popup will always handle the event in order to prevent it from propagating to the elements below.

As for the rendering, the popup can be set as modal, which means it will fill the entire overlay outside of the popup's boundaries with a given modal color and block any interaction with the rest of the user interface.

A popup can also be set to track a given point using a *PointTracker*.
There are a few predefined *PointTracker*s that, for example, allow to track a given point in the 3D-scene on the *RenderPanel* or to track the position of the *RenderPanel*'s center.
This allows popups to appear and stay next to elements in the 3D-scene, which is a requirement regarding **Intuitiveness** when displaying options related to that object.

In our scenario, this allows to show information about a checkpoint on the UMAD site and to create a measurement task of the given checkpoint when it is clicked.

Contrary to the user interface elements, popups also have an anchor point that can be set to one of the following values:

- AnchorTopLeft
- AnchorTopRight
- AnchorBottomRight
- AnchorBottomLeft
- AnchorCenter
- AnchorAuto

The anchor point determines where in the popup the point provided by the properties *top* and *left* – or the point returned by the *PointTracker* – is located.
Most of these are self-explanatory except for *AnchorAuto* which means that the popup will be displayed as it fits.
It will prefer to use the top left corner as the anchor but will adjust accordingly if it would not

fit on the screen entirely, and it is not limited to one of the five other anchor points.
The point has to be within the popup, though.
Meaning, if the popup is set to track a point in the 3D-scene and the camera moves far enough so that the point in the 3D-scene is not visible anymore, the popup will also be out of the screen either partly or entirely.

4.3.8 OverlayWrapperControl

At one point in the development of the overlay, all basic controls that were required had been developed and what was missing could be created using a combination of existing controls.
To make the creation of such controls that internally just wrap a couple of other controls easier and cleaner, the **OverlayWrapperControl** was created.
It basically has a child that is the actual control and forwards all events and property getters and setters to the child element.

The wrapping control can then simply inherit from this **OverlayWrapperControl**, pass it an element or in most cases an element tree and add functions that provide additional functionality based on the used controls.

For example, a panel that allowed to expand and collapse a section was needed.
To realize this functionality, the *OverlayExpandable* – see figure 4.5 – was created, which inherits from **OverlayWrapperControl** and internally creates an *OverlayGridLayout* with two rows and one column.

The first row contains an *OverlayButton*, containing an *OverlayStackPanel* with two children – an *OverlayText* for the open and close symbol and another *OverlayText* for the heading of the *OverlayExpandable*.
The second row contains the actual child that can be expanded and collapsed.

Finally, the *OverlayExpandable* connects to the signal that is emitted by the *OverlayButton* whenever it is clicked and switches its state between open and close.
If it's set to close, it simply sets the child's visible property to false.

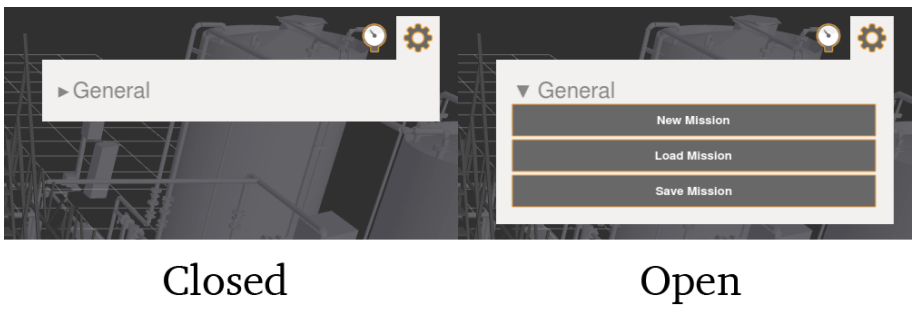


Figure 4.5.: An example of how the overlay expandable can be used.

4.4 Internal Representation of the Mission

Contrary to the structure in chapter 3, I will first introduce the internal representation of the mission before continuing with the visual representation.

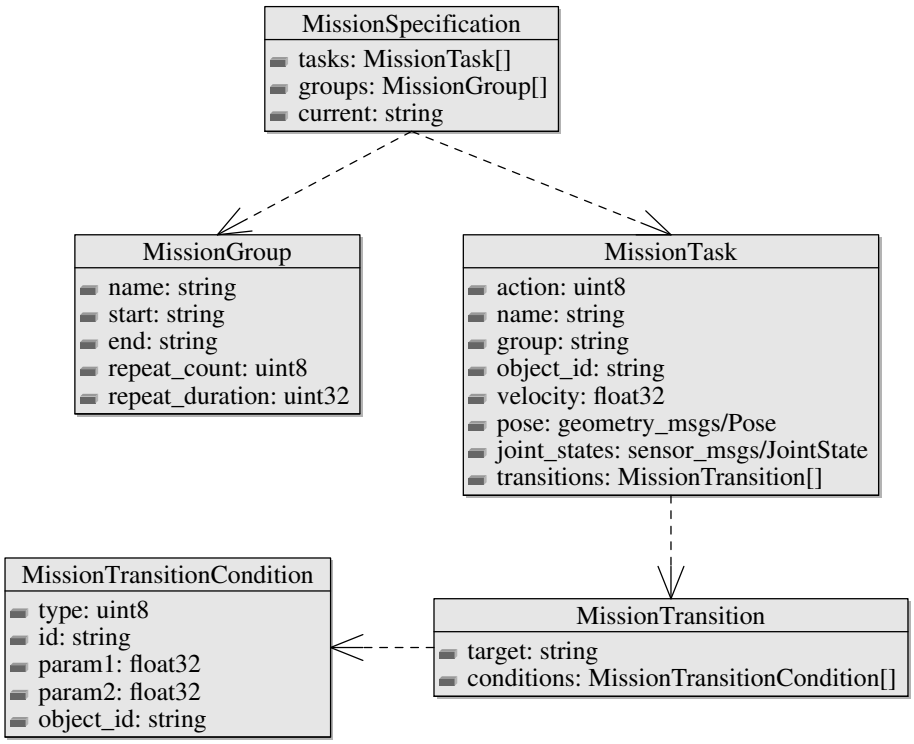


Figure 4.6.: The *ROS* message structure for the mission specification intermediate language.

The mission is internally represented as a set of *ROS* messages – as shown in figure 4.6.
Using *ROS* messages has a number of advantages.
For example, the existing *ROS* infrastructure can be used to send the mission to other nodes or to save and load it from the hard drive.
The biggest advantage of this approach, however, is that it is language independent.
There are *ROS* APIs to send, receive, store and load messages for most popular programming languages including C++ and Python.
Lastly, the biggest advantage over creating an own message channel is, of course, the *ROS* infrastructure being already widely used and very stable.

There are of course some restrictions such as missing support for recursive message definitions which means MissionGroup cannot have a member *subgroups* as a vector of MissionGroup.
Considering this is the only restriction imposed by using *ROS* messages, the concept was simply adapted and the hierarchy of groups is now expressed by using a slash '/' in the name.
For example, if *Beta* is a subgroup of *Alpha*, the name of *Beta* would be 'Alpha/Beta'.

The types used are special types for *ROS* messages, which are mapped to their language equivalent upon receiving or loading the message.
The messages depend on two other packages: *geometry_msgs* and *sensor_msgs*.

To communicate the target pose of a *WAYPOINT*, the *Pose* message from the *geometry_msgs* package is used. It consists of a point in 3D space and an orientation.

The joint_states were actually not part of the concept but added later on for the supervision user interface, that uses the same message infrastructure for semi-autonomous actions.
A *JointState* message consists of a list of strings with the names of the joints the *JointState* has values for, and float64 lists for the position, velocity, and effort of each joint.

4.5 ROS Nodes

4.5.1 Floor Publisher Node

The **Floor Publisher Node** is responsible for publishing the floor model.
To make it more flexible, the node can be configured using the parameters described in table 4.3.

Name	Description	Default Value
frame	The frame set in the header of each floor marker.	world
publish_topic	The topic the floor *MarkerArray* is published to.	argo_umad_floor_model
publish_current_topic	The topic where information about which floor is currently selected is published to.	argo_umad_floor_model
service_topic	As the name indicates, the service is advertised on this topic.	argo_umad_show_floor
floors_number	The number of floors.	6
scenario_name	The scenario determines which models to load. Currently, this can be either the UMAD site or a model of the local test site at TU Darmstadt.	umad_default

Table 4.3.: The parameters of the *Floor Publisher Node*

It publishes the floors up to the currently selected floor as a *visualization_msgs/MarkerArray* message, and an *argo_robot_control_ui_msgs/ArgoCurrentFloor* message containing the 1-based index of the current floor and a maximum z-value which tells *Subscribers* up to which height they should show objects.
If the highest floor is selected, the maximum z-value is set to the constant *FLT_MAX*.
Additionally, it also provides a service to change the selected floor.

4.5.2 Argo Interactive World Model Visualization Node

The **Argo Interactive World Model Visualization Node** retrieves all points-of-interest using the service *worldmodel/get_object_model* and subscribing to updates on the *worldmodel/object* topic.

The points-of-interest are parsed and filtered for checkpoints.
Checkpoints are added to a map, which maps them to their name.
This is done to allow the service advertised as *interactive_world_model/get_world_object* to quickly check if it contains a checkpoint with the name that was passed in the request and return the corresponding *hector_worldmodel_msgs::Object* message.

Furthermore, for the three different types of checkpoints – valve, dial gauge, and liquid level gauge – an *InteractiveMarker* containing a model of the checkpoint type, enclosed in a transparent box marker to increase the clickable area, is published – see figure 4.7.

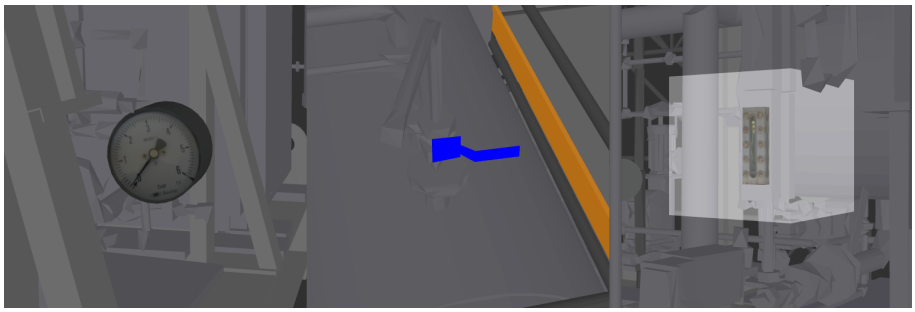


Figure 4.7.: The models for the different types of checkpoints on the site.
On the liquid level gauge, the mouse was hovered over the checkpoint to show the enclosing interactive button marker.

4.5.3 Back-end Planning Node

The **Back-end Planning Node** is an *ROS* node responsible for handling the mission.
It provides *Services* to:

- Create a new mission
- Save and load an existing mission
- Add, update or delete a task
- Add, update or delete a group
- Set the current task

Whenever the mission changes, it is published as an *argo_mission_msgs/MissionSpecification* on a latched topic[2].
Internally, the mission is handled by a class called **MissionPlan** which handles the *Service* calls and the generation of the *MissionSpecification* message from its stored data.

For better performance, pointers to the tasks and groups are stored redundantly.
Tasks use a double-linked list[3].
Groups are stored in a struct with a vector of the tasks contained in the group.
Additionally, the *MissionPlan* features an std::map for each the **Tasks** and the **Groups** which maps these values to their names as keys.

[2] A latched topic always stores the latest message and sends it to new subscribers.
[3] Every element of a double-linked list has a pointer to the next and the previous element in the list.

4.6 Visual Representation of the Mission

The visual representation of the planned mission was implemented as Display plugins for *RViz*. It is added as one Display called *PlanningDisplay* which internally creates instances of the other required Displays and handles the communication with the background node as depicted in figure 4.8.

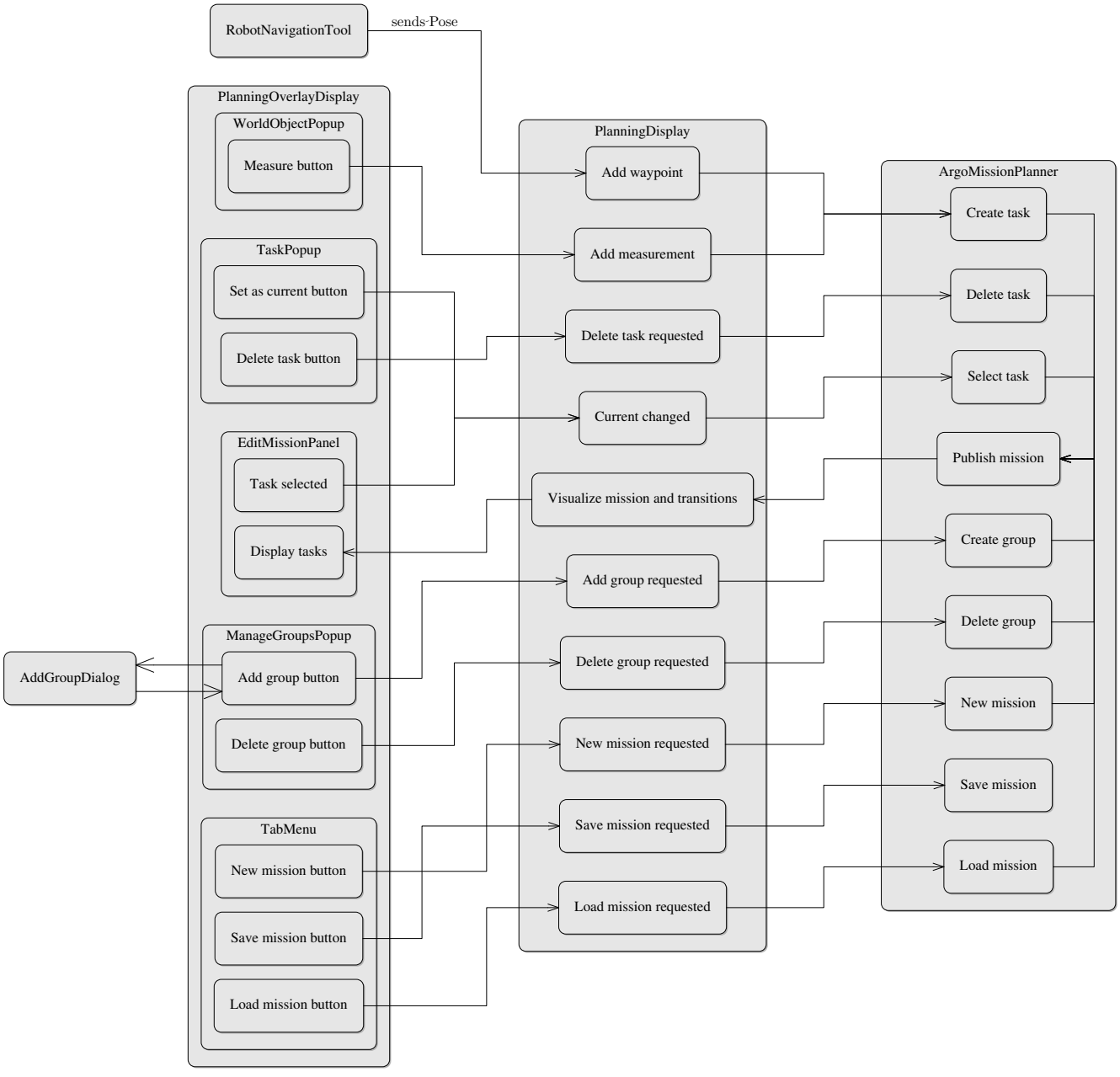


Figure 4.8.: The structure of the *PlanningDisplay* components.

Starting from the top to the bottom, the *PlanningDisplay* is responsible for combining the whole functionality into one Display.
It basically acts as a bridge between the back-end node and the different tools and overlay controls.

For example, if the *Navigation Tool* sends a *Pose*, the *Planning Display* wraps the pose into an *AddTask* request in order to add a new waypoint to the mission.
It is also responsible for displaying the current mission.
This is achieved by first iterating through all tasks and displaying the robot where it is expected to be when actually executing the mission.
When displaying a *WAYPOINT*, the display creates a *RobotStateVisualization*, provided by the *moveit*[11] package, moves it to the position set in the pose property and rotates it according to the orientation.
In the case of a *MEASURE*, the robot pose for the measurement has to be obtained from the *ROS* parameter server.
However, because the measurement position is relative to the checkpoint location, the location of the checkpoint has to be retrieved first using the *GetWorldObject* service provided by the *Argo Interactive World Model* – see 4.5.2.
To distinguish the robot that is currently selected, an alpha of 0.3 is applied to all other robots.

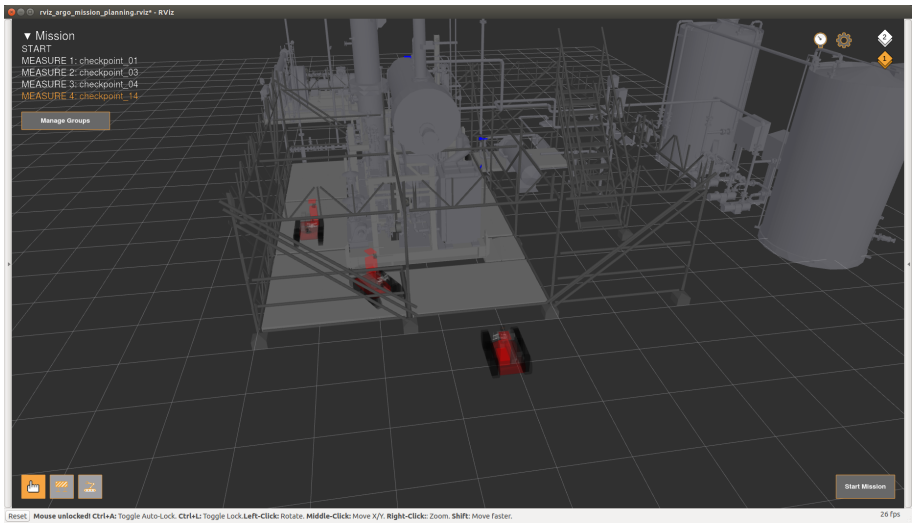


Figure 4.9.: Visual representation of the tasks.

As of now the robots are displayed but the user will not be able to tell which paths the robot is going to take, and more importantly in what order the robot will visit these points.
Hence, the next step is to visualize the transitions.
Visualizing the transitions is done in a separate loop after the robots are visualized to make sure the tasks already exist and there are no valid transitions that point to tasks that were not visualized yet and therefore there is no information available about where to draw the transition.
To give the best possible estimation of what the robot will actually do, the *Planning Display* uses a service provided by Mark Prediger's *Topo Planner*. The same planner is also used during the actual execution of the mission, therefore the displayed path should be reasonably accurate for simpler scenarios where the planned path is not in reality blocked by a previously unknown obstacle.
If the planner fails to find a path between the two tasks, as a fall-back an arrow pointing from the origin task to the transition target is drawn instead.

At this point, the mission visualization is very basic but complete.
What is still missing, is the interaction.
Interactivity is added using a default *RViz* Display plugin called *InteractiveMarkerDisplay* in combination with an *InteractiveMarkerServer*.
Using the *InteractiveMarkerServer* the *PlanningDisplay* publishes box markers for the robots, transitions are already displayed as markers, thus, they are simply wrapped by an *InteractiveMarker*.
The interactive markers are set to interact as buttons which means upon clicking on such a marker, a message is published to a topic the *PlanningDisplay* is subscribed to, the *PlanningDisplay* handles the click and triggers a corresponding action in the *PlanningOverlayDisplay*.

Finally, the *PlanningDisplay* also uses the *PlanningOverlayDisplay* which provides most of the overlay controls necessary for editing the mission.
It is responsible for displaying and handling almost all overlay controls shown in figure 4.10 except for the tool selection in the bottom left and the floor selection in the top right that are both separate Display plugins covered in section 4.2.
As in the concept, the mission is displayed in the top left. The currently selected task is highlighted in the accent color and can be chosen by clicking on the task's name.
The *Manage Groups* button opens a popup listing all groups and presents the option to add, edit or delete groups.
Adding and editing a group is done in a separate Qt dialog because the overlay that was developed specifically for this thesis does not support keyboard input.
Clicking on one of the buttons in the top right will open a pop-out menu providing options that are not frequently used.
On demand, for example, if the interactive marker for a robot is clicked, the overlay also shows popups that provide the option to edit or delete tasks and transitions.
These popups can track the position of a 3D point on the screen and stay at that point even if the camera moves.

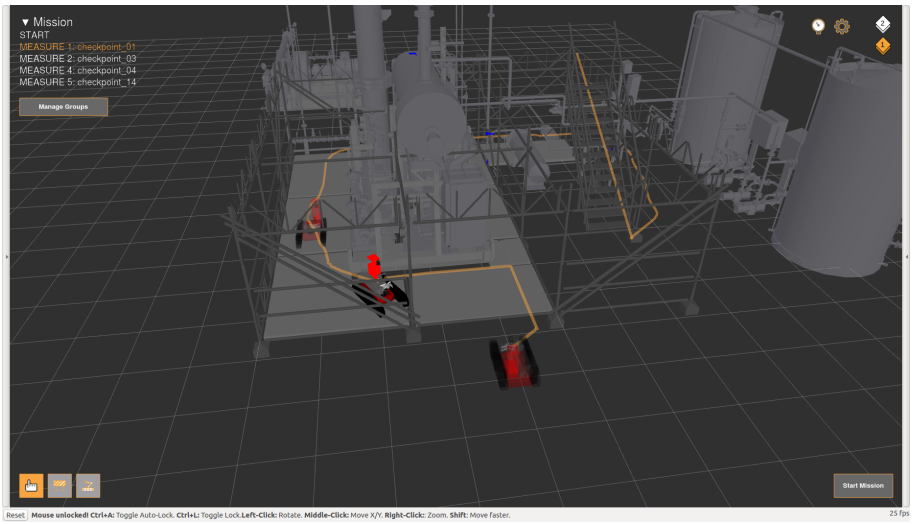


Figure 4.10.: The final visual representation of an example mission.

5 Evaluation

To evaluate the key aspects *Usability*, *Intuitiveness*, and *Responsiveness*, a user study was done in collaboration with Karim Barth.
We collaborated in this user study to also evaluate our common design and to get more representative results by having a larger base of study participants.
Usually, there are recommended practices and metrics for user interface evaluations[12].
However, considering this is a rather small study with only 15 participants due to the conduction of the study for a single participant taking between 30 minutes and 1 hour, the participants being mostly friends and family and the software used for executing the mission being in a pre-alpha state, the process was a bit simplified, and while the results provide a good indication, they are not representative.

It should also be noted that the operator in the ARGOS challenge is going to be someone with domain experience in oil and gas production sites.
This is not the case for any of the participants.
One participant, however, actually had experience in operating mobile robots.

5.1 Conduction

The study participants were instructed to create the mission according to the script given in figure 5.1.

Mission Protocol #2

1. Set a starting point in front of the UMAD site
2. Let the robot patrol the first floor in circles for 10 minutes
   - Hint: At least 3 robots are needed
3. Check the checkpoint #14
4. Add checkpoint #2 **after** checkpoint #14
5. Delete the measuring of checkpoint #2 and add the checkpoint #2 before checkpoint #14
6. Done!

Figure 5.1.: The script for the mission planning evaluation.

The mission was designed to cover all features currently available:

- Creation of waypoints
- Navigation to checkpoints using the checkpoints menu
- Creation of measurement tasks
- Creation of groups to repeat a set of tasks
- Insertion and deletion of tasks

To create the mission, the participant had to first set the *START* in front of the UMAD site model by using the *Robot Navigation Tool* (see 4.2.4).

Then, he or she had to place 4 *WAYPOINT* tasks in each corner of the bottom floor – 3 would suffice as well if positioned properly, otherwise, the robot might choose to drive backward.

To repeat these waypoints for 10 minutes, the user had to create a group by clicking on the 'Manage Groups' button in the top left, which opens a popup where the user had to click 'Add Group'.
This opens the dialog depicted in figure 5.2.

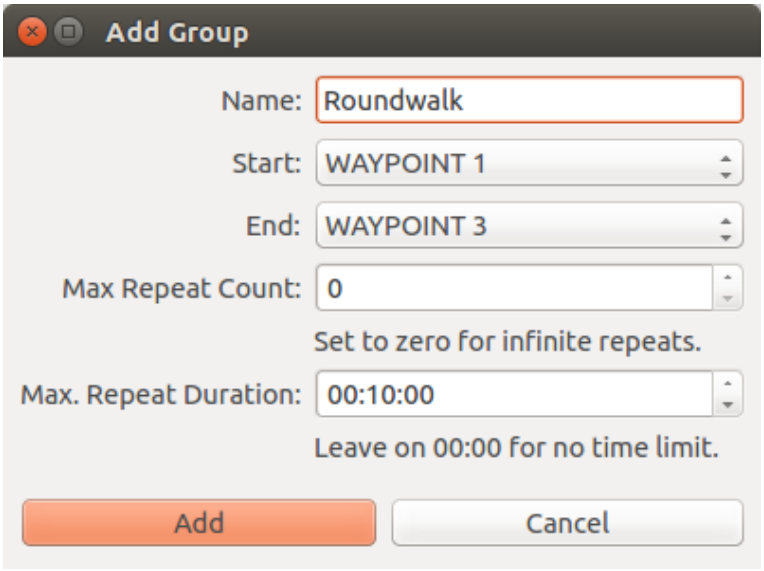


Figure 5.2.: The 'Create Group' dialog.

Since a name was not specified in the mission script, any name could be used.
As the start and end of the group, the first and the last waypoint on the site had to be chosen.
Then, the maximum duration had to be set to 10 minutes.
Finally, the group had to be added by clicking 'Add'.

Next, the participant had to navigate to checkpoint #14 using the checkpoint menu in the top right and add a *MEASUREMENT* task by first clicking on the valve checkpoint, which opens a popup, then, clicking on the 'Measure' button.
The same procedure had to be done for checkpoint #2.

To delete the measurement of checkpoint #2, the robot visualizing the *MEASUREMENT* task had to be clicked to open a popup with options regarding the task.
In the popup, the 'Delete task' button had to be clicked.

To add the checkpoint #2 before the measurement of checkpoint #14, the user could either select the last *WAYPOINT* task of the patrol group in the mission summary in the top left or by clicking on the robot visualizing the task and the 'Set as current' button in the opened popup.
Lastly, the measurement of checkpoint #2 could be added again as described earlier.

The resulting mission is displayed in figure 5.3.

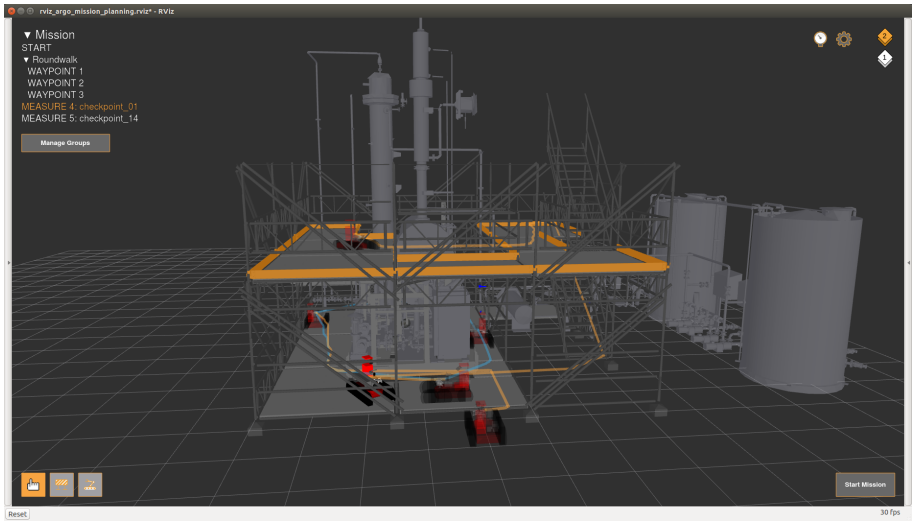


Figure 5.3.: A possible version of the mission created in the evaluation.

Afterward, they had to create and execute a simpler mission for the evaluation of the supervision user interface by Karim Barth[8].
Considering that the second part is not relevant for this thesis, it is not described in detail here.
Finally, every study participant had to fill out a form that can be found in the appendix [A.1].

5.2 Results

A summary of all results can be found in the appendix [A.2].

It took the 15 participants on average about 5min 28s to create the given mission.
The 7 participants who stated that they do play video games needed on average 50 seconds less (5min 02s) than the 8 participants who stated that they do **not** play video games.
While the sample size is too small to draw authoritative conclusions, it indicates that using video games as a source of inspiration made it easier for people with prior video game experience to orient themselves, but it is not needed, considering that the difference was just 50 seconds or around 15%.

All participants managed to create the given mission after a short introduction, which clearly shows that the concept of this user interface works as intended.
On a scale from 1 (Strongly disagree) to 4 (Strongly agree) the statement 'The user interface is intuitive / easy to use' was rated 3.2 on average, which is a good result.
Still, 20% slightly disagreed which indicates that there is room for improvement and ways to improve the **Intuitiveness** should be investigated.

Regarding *Responsiveness*, the participants rated the statement 'The planning interface feels responsive' on the previously mentioned scale 3.53 on average with no one disagreeing.

As for the integration of the planning and the supervision user interface, the design was deemed as consistent scoring a 3.47 whereas the statement 'The transition between planning and supervision is seamless' scored 3.33.

Additionally, the participants could make suggestions on how to improve the planning user interface.
Most of these suggestions were already planned for future development.
For example, preventing the setting of invalid waypoints with the *Robot Navigation Tool*, and better camera positions when looking at a checkpoint using the checkpoint menu.
However, there was also very valuable new feedback like the option to reorder tasks in the planned mission using drag & drop.

6 Conclusion and Outlook

In this thesis, a more intuitive user interface for the planning of missions in the context of the ARGOS challenge was introduced.

Aside from the *RViz* plugins that were developed specifically for this user interface, a general purpose overlay with mouse support was created.
For the specification of mission plans and to provide an abstraction from the FlexBe automaton, an intermediate language for missions in the form of *ROS* messages was developed.
To navigate within the 3D-scene, a *ViewController* that allows movement in the 3D-scene using cursor locking and screen-edge camera movement or, alternatively, using the arrow keys was developed.

As the user study in chapter 5 indicates, the user interface works as intended.
After a short introduction of only around 15 minutes as opposed to the 30 minutes given in the 3rd competition, it was possible for users without detailed knowledge about the used robot control software to create a mission in a relatively short amount of time.

As a next step, to improve the **Usability** and **Intuitiveness** of this user interface, the feedback collected in the evaluation will be implemented.
In order to allow the creation of more complex missions, conditional transitions will be added.
Furthermore, the *Place Obstacle Tool* will be updated once the new obstacle server is available.

Additionally, the integration of the planning and the supervision user interface during execution of the mission will be improved.
For example, different approaches to better integrate information about the current state of the robot and the execution state of the mission in the planning user interface will be investigated.

In conclusion, the user interface presented in this thesis provides the required capabilities for a mission planner allowing for the intuitive and fast creation of industrial inspection mission plans.

Bibliography

[1] Philipp Schillinger. “An Approach for Runtime-Modifiable Behavior Control of Humanoid Rescue Robots”. MA thesis. Technische Universitaet Darmstadt, Department of Computer Science (SIM), 2015.

[2] Stefan Kohlbrecher and Oskar von Stryk. “From RoboCup Rescue to Supervised Autonomous Mobile Robots for Remote Inspection of Industrial Plants”. In: *KI - Künstliche Intelligenz* 30.3 (2016), pp. 311–314. ISSN: 1610-1987. DOI: 10.1007/s13218-016-0446-8.

[3] TOTAL ANR. “ARGOS Challenge - Rules of the 3rd competition”.

[4] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA Workshop on Open Source Software*. 2009.

[5] Hyeong Ryeol Kam et al. “RViz: A Toolkit for Real Domain Data Visualization”. In: *Telecommun. Syst.* 60.2 (Oct. 2015), pp. 337–345. ISSN: 1018-4864. DOI: 10.1007/s11235-015-0034-5.

[6] *OGRE - Open Source 3D Graphics Engine*. URL: http://www.ogre3d.org/ (visited on 11/13/2016).

[7] Péter Frankhauser. “ANYmal at the ARGOS Challenge”. In: Team LIO - ETH Z"urich. URL: http://roscon.ros.org/2016/presentations/ROSCon%202016%20-%20ANYmal.pdf.

[8] Karim Barth. “Development of a User Interface for Supervised Inspection using mobile Robots in challenging Environments”. BA Thesis. Technische Universitaet Darmstadt, Department of Computer Science (SIM), 2016.

[9] Antoine de Saint-Exupéry. *Terre des Hommes*. 1939.

[10] Adam Leeper. URL: https://github.com/ros-visualization/rviz_animated_view_controller (visited on 11/21/2016).

[11] Sachin Chitta, Ioan Sucan, and Steve Cousins. “MoveIt!” In: *IEEE robotics & automation magazine* 19 (2012), pp. 18–19. ISSN: 1070-9932. DOI: 10.1109/MRA.2011.2181749.

[12] Melody Yvette Ivory. “An Empirical Foundation for Automated Web Interface Evaluation”. PhD thesis. UC Berkeley, 2011.

A Appendix

List of Figures

1.1. An example of a mission planned in FlexBe. 1
1.2. The UMAD site where the competition is held. 2
1.3. The functional categories, taken from the 'Rules of the 3rd competition'[3] 3
1.4. The script of the 1st mission of the 3rd competition. 4
1.5. The different kinds of plugins in *RViz* 6

2.1. Mission Creation for Team LIO's ANYmal. 9

3.1. Mission planning in strategic video games (Planetary Annihilation). 10
3.2. An exemplary mission flow. 11
3.3. The message structure for the intermediate language. 12
3.4. A concept of the user interface. 14

4.1. Part of the structure of the overlay's elements. 20
4.2. How the effective value of the pen and background property is determined. 21
4.3. The effect of margin and padding on the layout.
The black rectangle marks the border of the element. 23
4.4. An example of how the different states can be used to visually represent an element's state. 23
4.5. An example of how the overlay expandable can be used. 27
4.6. The *ROS* message structure for the mission specification intermediate language. . 28
4.7. The models for the different types of checkpoints on the site.
On the liquid level gauge, the mouse was hovered over the checkpoint to show the enclosing interactive button marker. 30
4.8. The structure of the *PlanningDisplay* components. 31
4.9. Visual representation of the tasks. 32
4.10. The final visual representation of an example mission. 33

5.1. The script for the mission planning evaluation. 34
5.2. The 'Create Group' dialog. 35
5.3. A possible version of the mission created in the evaluation. 36

List of Tables

3.1. The different condition types. 13

4.1. The different lock modes supported by the **Mouse View Controller**. 16
4.2. The different units of the row and column definitions. 25
4.3. The parameters of the *Floor Publisher Node* 29

A.1 Survey

Argo Mission Planning and Supervision Survey

* Erforderlich

Mission Detail

To be filled by supervisor.

1. **Planning Time** *

   Beispiel: 4:03:32 (4 Stunden, 3 Minuten, 32 Sekunden)

2. **Supervision Time Start (Simulation)** *

   .....

   Beispiel: 4:03:32 (4 Stunden, 3 Minuten, 32 Sekunden)

3. **Supervision Time End (Simulation)** *

   .....

   Beispiel: 4:03:32 (4 Stunden, 3 Minuten, 32 Sekunden)

4. **Supervision Time Start (Realtime)** *

   .....

   Beispiel: 4:03:32 (4 Stunden, 3 Minuten, 32 Sekunden)

5. **Supervision Time End (Realtime)** *

   .....

   Beispiel: 4:03:32 (4 Stunden, 3 Minuten, 32 Sekunden)

6. **Result** *

   Markieren Sie nur ein Oval.

   - ◯ Success
   - ◯ Failed
   - ◯ Techinical difficulties

7. **Additional Info**

   .....

   .....

   .....

   .....

General

To be filled by user

8. **Please choose your gender:**

   Markieren Sie nur ein Oval.

   - ◯ Male
   - ◯ Female

9. **How old are you?**
Markieren Sie nur ein Oval.

- under 18
- 18-25
- 25-35
- 35-55
- 55 or older

10. **Do you have experience in operating mobile robots?** *
Markieren Sie nur ein Oval.

- Yes
- No

11. **Do you play video games in particular the genres Strategy or Multiplayer Online Battle Arena?** *
Markieren Sie nur ein Oval.

- Yes
- No

12. **The time of the introduction was sufficient** *
Markieren Sie nur ein Oval.

	1	2	3	4	
Strongly disagree					Strongly agree

13. **The transition between planning and supervision is seamless** *
Markieren Sie nur ein Oval.

	1	2	3	4	
Strongly disagree					Strongly agree

14. **The design of the planning and the supervision user interface feels consistent** *
Markieren Sie nur ein Oval.

	1	2	3	4	
Strongly disagree					Strongly agree

Mission Planning

15. **The user interface is intuitive / easy to use** *
Markieren Sie nur ein Oval.

	1	2	3	4	
Strongly disagree					Strongly agree

16. **The planning interface feels responsive** *
Markieren Sie nur ein Oval.

	1	2	3	4	
Strongly disagree	◯	◯	◯	◯	Strongly agree

17. **I had no problems creating the given mission** *
Markieren Sie nur ein Oval.

	1	2	3	4	
Strongly disagree	◯	◯	◯	◯	Strongly agree

Supervision

18. **The user interface is intuitive / easy to use** *
Markieren Sie nur ein Oval.

	1	2	3	4	
Strongly disagree	◯	◯	◯	◯	Strongly agree

19. **The meaning of the different operation modes is comprehensive and clear** *
Markieren Sie nur ein Oval.

	1	2	3	4	
Strongly disagree	◯	◯	◯	◯	Strongly agree

20. **Was is obvious which operation mode you where in?** *
Markieren Sie nur ein Oval.

- ◯ Yes
- ◯ No

21. **Did the different accent colors facilitate your orientation?** *
Markieren Sie nur ein Oval.

- ◯ Yes
- ◯ No

Additional

22. **Do you have suggestions for improvement?**

.....

.....

.....

.....

A.2 Survey Results

15 Antworten

Zusammenfassung

Mission Detail

Planning Time

00:__:__							
00:08:14	00:07:19	00:03:52	00:07:49	00:04:29	00:03:03	00:06:05	00:04:31
00:00:47	00:04:40	00:08:28	00:07:33	00:06:56	00:04:42	00:03:37	

Supervision Time Start (Simulation)

00:__:__							
00:07:25	00:00:13	00:08:07	00:09:40	00:07:34	00:00:33	00:05:35	00:07:37
00:01:49	00:11:32	00:11:09	00:05:20	00:00:52	00:04:18	00:03:20	

Supervision Time End (Simulation)

00:__:__							
00:20:56	00:10:49	00:16:00	00:24:06	00:12:59	00:10:03	00:16:41	00:18:32
00:14:17	00:23:21	00:20:47	00:13:40	00:10:52	00:14:11	00:14:29	

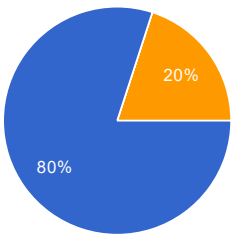
Supervision Time Start (Realtime)

00:__:__							
00:14:02	00:00:57	00:14:38	00:18:35	00:13:35	00:00:59	00:09:49	00:13:39
00:03:15	00:21:54	00:20:09	00:23:40	00:03:07	00:08:12	00:06:36	

Supervision Time End (Realtime)

00:__:__							
00:41:12	00:43:21	00:29:54	00:48:57	00:33:30	00:19:26	00:31:10	00:34:30
00:21:40	00:46:06	00:38:56	00:45:30	00:43:06	00:28:15	00:29:00	

Result



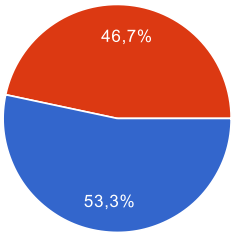
Success	**12**	80 %
Failed	**0**	0 %
Techinical difficulties	**3**	20 %

Additional Info

CP 14 konnte nicht gemessen werden
An der Treppe hängen geblieben
Roboter ist am Hindernis hängen geblieben, Verzögerung durch Desktopübertragung

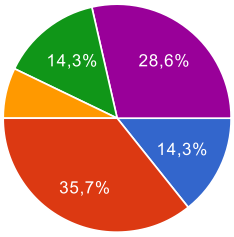
General

Please choose your gender:



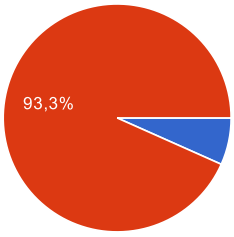
Male	**8**	53.3 %
Female	**7**	46.7 %

How old are you?



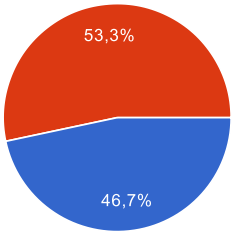
under 18	**2**	14.3 %
18-25	**5**	35.7 %
25-35	**1**	7.1 %
35-55	**2**	14.3 %
55 or older	**4**	28.6 %

Do you have experience in operating mobile robots?



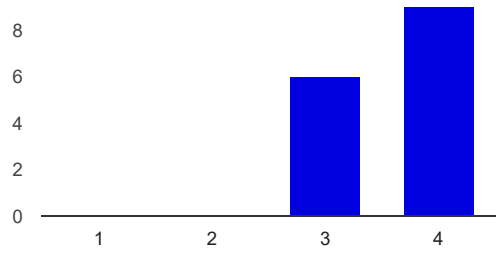
Yes	**1**	6.7 %
No	**14**	93.3 %

Do you play video games in particular the genres Strategy or Multiplayer Online Battle Arena?



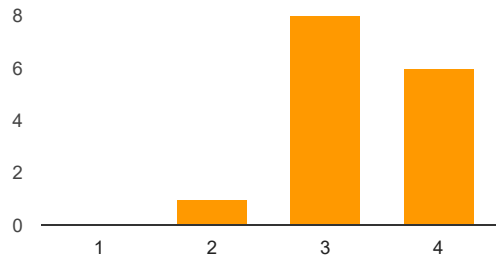
Yes	**7**	46.7 %
No	**8**	53.3 %

The time of the introduction was sufficient



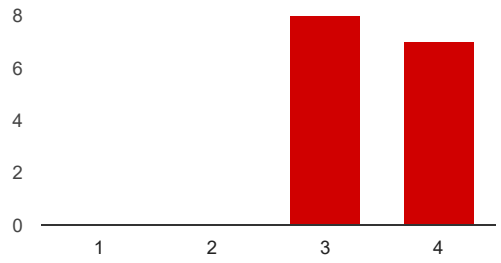
Strongly disagree:	1	**0**	0 %
	2	**0**	0 %
	3	**6**	40 %
Strongly agree:	4	**9**	60 %

The transition between planning and supervision is seamless



Strongly disagree:	1	**0**	0 %
	2	**1**	6.7 %
	3	**8**	53.3 %
Strongly agree:	4	**6**	40 %

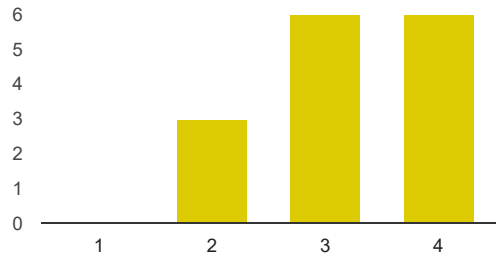
The design of the planning and the supervision user interface feels consistent



Strongly disagree:	1	**0**	0 %
	2	**0**	0 %
	3	**8**	53.3 %
Strongly agree:	4	**7**	46.7 %

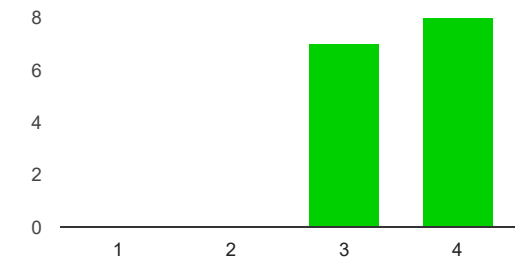
Mission Planning

The user interface is intuitive / easy to use



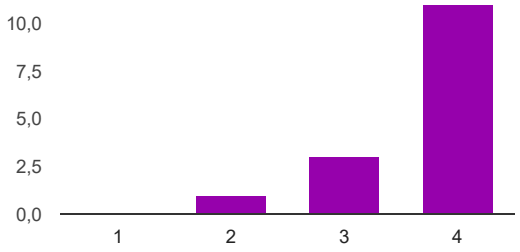
Strongly disagree:	1	**0**	0 %
	2	**3**	20 %
	3	**6**	40 %
Strongly agree:	4	**6**	40 %

The planning interface feels responsive



Strongly disagree:	1	**0**	0 %
	2	**0**	0 %
	3	**7**	46.7 %
Strongly agree:	4	**8**	53.3 %

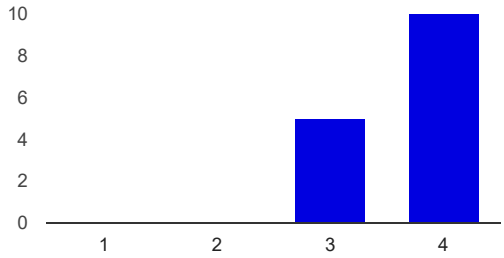
I had no problems creating the given mission



Strongly disagree:	1	**0**	0 %
	2	**1**	6.7 %
	3	**3**	20 %
Strongly agree:	4	**11**	73.3 %

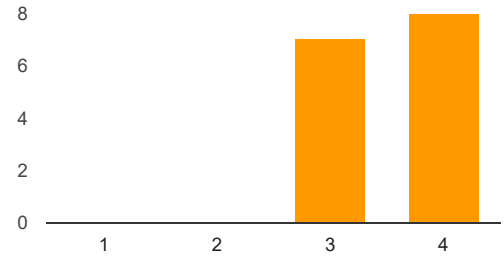
Supervision

The user interface is intuitive / easy to use



Strongly disagree:	1	**0**	0 %
	2	**0**	0 %
	3	**5**	33.3 %
Strongly agree:	4	**10**	66.7 %

The meaning of the different operation modes is comprehensive and clear

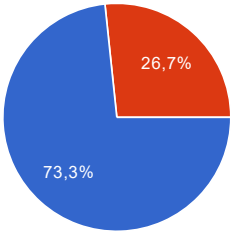


Strongly disagree:	1	**0**	0 %
	2	**0**	0 %
	3	**7**	46.7 %
Strongly agree:	4	**8**	53.3 %

Was is obvious which operation mode you where in?

Yes	**15**	100 %
No	**0**	0 %

Did the different accent colors facilitate your orientation?



Yes	**11**	73.3 %
No	**4**	26.7 %

Additional

Do you have suggestions for improvement?

Navigation Tool: Ungültige Eingaben sollten unterbunden werden.
Teilweise schlechte Einsicht auf die site.
Checkpoint Reihenfolge in der Planung löschen oder per Drag and Drop verschieben können; Eine Kamera, die dem Roboter automatisch folgt; Im Supervisory Modus bei der manuellen Navigation (Navigation Tool) eine Markierung setzen, damit ersichtlich ist an welche Stelle der Roboter fahren soll; Bei den einzelnen Modi bei langem daraufbleiben der Maus eine kleine Infobox einblenden, was die einzelnen Modi nochmal genau machen; Der Roboter sollte den Arm nach jeder Aktion automatisch wieder einfahren, egal ob eine Aktion erfolgreich war oder nicht; Kameras am Roboter auf einer horizontalen Ebene belassen (nicht schief, quer etc.); Bei den unterschiedlichen Kameraperspektiven (wenn möglich) Gegenstände (Bsp. Stangen) im Sichtfeld verschwinden lassen bzw. durchsichtig machen; Ich finde es umständlich das Tool für die Positionierung des Roboters (Navigation Tool/Waypoint Tool) für jeden einzelnen Wegpunkt neu auswählen zu müssen. Besser: durch das Klicken mit der Maus (linke Maustaste) Waypoints setzen und automatisch verbinden; Exklusives Anzeigen der Stockwerke mit Eingabeoptionen nur auf das ausgwählte Stpckwerk beschränkt (damit es nicht passieren kann, dass ich mich im zweiten Stockwerk befinde und den Roboter ausversehen ins erste Stockwerk setze); Bei der Auswahl unterschiedlicher Tools die Auswahl mit einem Rechtsklick wieder aufheben; Im Planungsmodus das Fenster vom Checkpoint durch einen Random-Klick in die Gegend wieder schließen oder die Checkpoints nicht als eigenes Fenster, sondern beispielweise unten als dauerhaft angezeigte Leiste einblenden.
Im Planmodus sollten die Einzelschritte einfacher zu bearbeiten und verschieben sein. Die Fenster im Planmodus sollten sich schliessen wenn man in den Hintergrund klickt. Den Roboterplazierknopf zum Navigieren ist leicht verwirrend. Eventuell vielleicht eine Kurzanleitung in die Fenster setzen(Aufrufbar über Hilfe-Knopf). Kamera sollte Roboter verfolgen können. Stockwerk sollen wechseln wenn Roboter auf

ein anderes Stockwerk fährt. Was man in welchem Modus tun kann sollte eindeutiger sein, ansonsten wirklich intuitiv gelöst. Sehr einfache und verständliche Oberfläche!
CP teilweise nicht vollständig sichtbar.
- Kontrast der CP Objekte teilweise zu niedrig. Blaue valve auf blauen Hintergrund. Graue auf grau; - Eventuell CP Nummer am Objekt direkt anzeigen
Missionsplanung: Beim Anklicken einiger Checkpoints im Checkpointmenü ist der Kamerawinkel ungünstig gewählt. Hat man eine Mission an einen bestimmten Punkt gelegt lässt sich diese nicht mehr verschieben. Man muss sie erst löschen und neu einfügen. Missionsausführung: Da der Roboter rückwärts die Stufen hinauf fährt muss man sich erst einmal mit der Kamerasetzung (hinter, vor, links, rechts, über) auseinandersetzen, um zu wissen welche Position die ist, die man haben möchte. Hat man die Kamera an einen Punkt fixiert (z.B. hinter den Roboter) folgt diese ihm nicht. Das manuelle Editieren funktioniert nur mit dem Klicken auf das - bzw. das + Symbol. Vielleicht wäre die Eingabe in einem Feld mittels Zifferntasten eine Option?
Beschriftung der Checkpoints in der 3D-Szene; Farbige Hervorhebung der Checkpoints in der 3D-Szene nachdem einer in dem Auswahlmenü ausgewählt wurde; Direktes Auswählen der Checkpoints durch Auswahl durch das Auswahlmenü (statt nur Sprung zu dem Checkpoint in der 3D-Szene); 3D-Modelle in der 3D-Szene, die das Blickfeld des Users stören (z.B. die Stangen) entweder entfernen oder eine Möglichkeit ergänzen, diese für den Nutzer unsichtbar zu machen
- Missionsplanung/ -umplanung über drag and drop - Sichtbarkeit der Instrumente nicht immer gegeben - Kennzeichnung der Instrumente - Bedienfelder sollten immer komplett sichtbar sein - Roboter sollte immer im Bild bleiben, oder in Draufsicht springen
Roboter sollte immer mit der Kamera verbunden sein Farbgestaltung könnte verbessert werden
Maybe a possibility to iterate over the waypoints using e.g alt + mouse wheel.