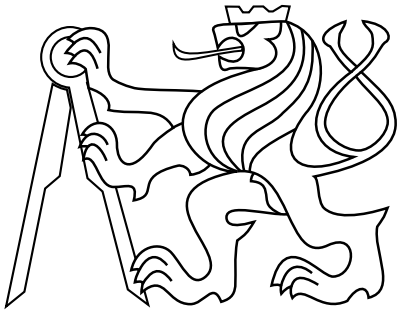
Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Control Engineering



Mobile testing robot

Master's Thesis

Bc. Jan Pražák

Prague, May 2024

Study programme: Open Informatics
Branch of study: Computer Engineering

Supervisor: prof. Dr. Ing. Zdeněk Hanzálek

Acknowledgments

I would like to express my deepest gratitude to my supervisor, prof. Dr. Ing. Zdeněk Hanzálek, for his invaluable guidance and insightful advice throughout the development of this thesis. His expertise and dedication have been instrumental in shaping this work.

I extend my heartfelt thanks to my colleagues, especially Ing. Roman Zima and Bc. Roman Šíp, for their collaboration on the project. Without them, the successful realisation of this project would not have been possible.

I am also profoundly grateful to my friends and family for their continuous support, understanding, and encouragement. Their belief in me has been a constant source of motivation during the challenging moments of my work.

Lastly, I would like to acknowledge everyone who has contributed to my academic journey and the completion of this thesis, directly or indirectly. Your support has been greatly appreciated.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Pražák Jan** Personal ID number: **492070**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Open Informatics**

Specialisation: **Computer Engineering**

II. Master’s thesis details

Master’s thesis title in English:

Mobile testing robot

Master’s thesis title in Czech:

Mobilní testovací robot

Guidelines:

Choose a suitable means of realization of a user and machine interface for a mobile testing robot connected to the system under test (SUT - presumably specific functions of a series production car).
Design a GUI, central management ROS node, communication with SUT, and peripheral control system.
Implement the SW components mentioned above, suitable test procedures, and evaluate the performance and reliability of the system.

Bibliography / sources:

Takaya, Kenta, et al. "Simulation environment for mobile robots testing using ROS and Gazebo." 2016 20th International Conference on System
Theory, Control and Computing (ICSTCC). IEEE, 2016.
Bouchier, Paul. "Embedded ros [ros topics]." IEEE Robotics & Automation Magazine 20.2 (2013): 17-19.
Currier, Chris. "Protocol buffers." Mobile Forensics–The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices. Cham: Springer International Publishing, 2022. 223-260.
Sobotka, Jan, Jirí Novák, and Jirí Pinkava. "Rapid
Prototyping of Vehicle Software Defined Functions."
26th IMEKO TC4 Symposium and 24th International Workshop on ADC and DAC Modelling and Testing (IWADC), 2023.

Name and workplace of master’s thesis supervisor:

prof. Dr. Ing. Zdeněk Hanzálek Department of Control Engineering FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **09.01.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until:
by the end of summer semester 2024/2025

prof. Dr. Ing. Zdeněk Hanzálek Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Declaration

I declare that presented work was developed independently, and that I have listed all sources of information used within, in accordance with the Methodical instructions for observing ethical principles in preparation of university theses.

Date ....

Abstract

Automated testing processes are integral to modern project workflows. In this thesis, we present the design and implementation of a mobile testing robot suitable for various scenarios, with an emphasis on use in the automotive industry. The robot is based on the Jackal platform and uses the Robot Operating System (ROS) for inter-node communication and integration of packages. A graphical application, developed using C# and Avalonia UI, facilitates remote operation of the robot via Wi-Fi, employing Protocol Buffers (protobuf) for efficient data exchange. Controller Area Network (CAN) is utilised to monitor the system under test. Finally, we outline several methods used to verify the correctness of the developed system and conduct real-world experiments to assess its performance and reliability.

Keywords Robot, Test Automation, Robot Operating System, Controller Area Network, .NET

Abstrakt

Automatické testování procesů je nedílnou součástí práce na moderních projektech. V této práci představíme design a implementaci mobilního testovacího robota vhodného pro různé scénáře s důrazem na využití v automobilovém průmyslu. Robot je vytvořen na platformě Jackal a využívá Robot Operating System (ROS) pro komunikaci mezi nody a integraci balíčků. Grafická aplikace vyvinutá v C# a Avalonia UI zprostředkovává vzdálené ovládání robota přes Wi-Fi, využívaje Protocol Buffers (protobuf) pro efektivní výměnu dat. Controller Area Network (CAN) je využit pro monitorování testovaného systému. Nakonec nastíníme metody použité k ověření správnosti vyvinutého systému a provedeme reálné experimenty pro vyhodnocení jeho výkonnosti a spolehlivosti.

Klíčová slova Robot, Automatizace Testování, Robot Operating System, Controller Area Network, .NET

Abbreviations

API Application Programming Interface

LED Light-Emitting Diode

SUT System Under Test

GPS Global Positioning System

IMU Inertial Measurement Unit

LiDAR Light Detection and Ranging

UI User Interface

GUI Graphical User Interface

TCP Transmission Control Protocol

IP Internet Protocol

ROS Robot Operating System

protobuf Protocol Buffers

CIL Common Intermediate Language

CLI Common Language Infrastructure

CLR Common Language Runtime

CPU Central Processing Unit

XML Extensible Markup Language

XAML Extensible Application Markup Language

AXAML Avalonia XAML

WinForms Windows Forms

WPF Windows Presentation Foundation

MAUI Multi-platform App UI

CSS Cascading Style Sheets

SVG Scalable Vector Graphics

CSV Comma-Separated Values

PNG Portable Network Graphics

JSON JavaScript Object Notation

CAN Controller Area Network

USB Universal Serial Bus

SLAM Simultaneous Localization And Mapping

Contents

1 Introduction **1**
1.1 Definitions . . . 2

2 Application **3**
2.1 Autolock scenario . . . 3
2.2 Key press scenario . . . 4

3 Design **9**
3.1 Basic actions . . . 9
3.2 Operating software to Robot communication . . . 10
3.3 Architecture . . . 10
3.4 Output . . . 13

4 Implementation **15**
4.1 Operating software . . . 15
4.1.1 The C# programming language . . . 15
4.1.2 Graphical user interface . . . 16
4.1.2.1 UI frameworks for .NET . . . 16
4.1.2.2 Connection window . . . 17
4.1.2.3 Main window . . . 21
4.1.3 Robot navigation . . . 32
4.1.4 Robot communication . . . 33
4.1.4.1 Outbound communication . . . 33
4.1.4.2 Inbound communication . . . 36
4.1.4.3 Serialisation format . . . 36
4.1.5 Car communication . . . 36
4.2 Robot . . . 37
4.2.1 Hardware . . . 37
4.2.2 Software . . . 39
4.2.2.1 Robot Operating System . . . 39
4.2.2.2 State machine . . . 40
4.2.2.3 Operating software communication . . . 42
4.2.2.4 Peripherals microcontroller . . . 42

5 Testing and evaluation **43**
5.1 Unit testing . . . 44
5.2 Mocking . . . 46
5.3 Connection server . . . 46
5.4 Integration testing . . . 48

5.5 Tools for manual testing and development 48
5.5.1 Robot substitution 50
5.5.2 Car substitution 50
5.5.3 Gazebo 50
5.5.4 RViz 51
5.6 Real-world experiments 51
5.7 Evaluation 52

6 Conclusion **53**

7 References **55**

Chapter 1

Introduction

A robot is an autonomous system that performs specific actions in the real world. A testing robot is a particular use of a robot whose actions interact with another system, called the System Under Test (SUT). The effect of the robot's actions on the SUT is observed to determine whether the SUT functions as intended. A mobile testing robot is further capable of movement in the real world.

In this thesis, the testing robot will be a four-wheeled ground vehicle. Several peripherals will be attached to the testing robot, consisting of motors and Light-Emitting Diodes (LEDs). The motors allow the robot to interact with the SUT, while the LEDs signalise the current state of the robot to the user (i.e., the human operator running the test). Additionally, a computer running specialised software will be directly connected to the SUT to observe the changes induced by the robot's actions. Based on these observations, the computer will use a wireless connection to guide the robot through the test.

The testing robot's motion-planning capabilities do not need to be overly complex, as the area in which the robot moves should be free of obstacles; however, since a person could enter the testing area, the robot is expected to stop to prevent a collision if a person or another object is detected in its path. For details about more advanced approaches to motion- and trajectory-planning, see [1], [2], [3], [4], and [5].

The general idea of the test scenario discussed in this thesis will consist of a static object positioned in the centre of a two-dimensional coordinate system and another object carried by the testing robot. The SUT will be the combination of the static object, the object carried by the robot, and the interaction between these two objects, where the interaction can be the result of an autonomous operation of the static object and the carried object, or triggered by the robot's motors' actions on the carried object. The testing robot approaches the static object from various positions to determine how the distance and angle between the static object and the carried object influence the interaction.

The test result will be a list of positions at which the interaction has been deemed to function correctly and reliably. The data obtained from the test can be used by the manufacturer of the SUT to discover system flaws, improve its performance, and verify the correctness of the system before shipping to customers.

To illustrate the use of the proposed testing robot, this thesis will focus on a specific application of the testing robot to test the communication between a remote car key and its receiver located inside the car. Chapter 2 presents the application in more detail. Other possible applications, which will not be discussed further in this thesis, include

- testing object recognition for a fixed object from different angles and distances,
- testing sound distribution around a speaker, or

- testing wireless communication over specific technologies, e.g. Bluetooth[1] or Wi-Fi[2].

1.1 Definitions

Several terms are defined and used in this thesis to avoid ambiguity. The complete list of such terms is in Table 1.1. The terms are always written with the first letter capitalised.

Robot	the presented mobile testing robot
Remote key	the remote car key that is a part of the SUT
Receiver	the receiver (or multiple receivers) inside the car that is a part of the SUT
Operating software	the software running on the computer directly connected to the SUT
Operator	the human operator running the test
Car	the car that is a part of the SUT

Table 1.1: List of terms defined and used in this thesis

[1] https://en.wikipedia.org/wiki/Bluetooth
[2] https://en.wikipedia.org/wiki/IEEE_802.11

Chapter 2

Application

The application discussed in this thesis serves as an example of possible real-world use of the mobile testing robot and the Operating software. The general idea of the application is to find the limit range at which communication between the Car and the Remote key functions reliably. The Remote key is tested from various positions around the Car as the communication might be affected by the angle between the transmitter and the receiver and by obstacles in the path of the sent signals, most notably by the metal body of the Car itself.

Nowadays, apart from a remote key that opens a car when a button is pressed, some cars also support keys that do not require a button press from the car's user to unlock the car; see [6]. Instead, the car is unlocked automatically when the remote key is located in the car's vicinity, and then the car locks itself once the remote key is removed.

To showcase how the Robot and the Operating software can be used in different scenarios, the designed system supports both types of keys. The scenario in which the Car's locks react automatically to the presence of the Remote key is called the *Autolock* scenario in this thesis and is described in Section 2.1. The second test scenario in which the Remote key requires a button press to unlock the Car is called the *Key press* scenario and is further detailed in Section 2.2.

As mentioned in Chapter 1, a test scenario consists of a static object positioned in the centre of the coordinate system and another object that the Robot carries. In both test scenarios, we assume the static object to be the Car and the carried object to be the Remote key. The Robot moves around the Car and attempts to lock/unlock the Car. At the same time, the Operating software expects to receive a signal from the Receiver indicating whether the lock/unlock has been successful. The lock/unlock attempt is repeated at various positions so that the Operator can better understand the limits in the communication between the Remote key and the Receiver.

2.1 Autolock scenario

The *Autolock* scenario tests the Car's ability to unlock and lock itself when the Remote key enters and leaves a small area around it. The test output is a set of points approximating the area's border.

It might seem sufficient to measure only the points at which the Car either locks or unlocks itself; however, generally, the locking/unlocking mechanism could be composed of two separate areas. When the Remote key enters the first area closer to the Car, the Car is unlocked. Then, once the Remote key leaves the second area, further from the Car, the Car becomes locked.

If only one area was to be used, the behaviour when the Remote key was located at the area's border would be indeterminate and might confuse the Car's user. For this reason, using the two areas mentioned above may be preferable as it improves the system's stability.

Therefore, the *Autolock* test scenario measures both the points that lock and unlock the Car. Note that if the Car internally only used one area, measuring the two separate areas would not devalue the final results, as it is a more general approach.

In the *Autolock* test scenario, the unlock/lock points are measured with the Robot moving to/from each side and corner of the Car. For each side of the Car, the axis orthogonal to the side's centre is tested, as well as every axis that is parallel to the centre axis and is located $k \cdot d_{mes}$ from the centre axis, for each $k \in \mathbb{N}$ while the axis does not exceed the boundaries of the Car, and for a $d_{mes} \in \mathbb{R}^+$, which is a parameter of the test.

The Robot starts a measurement on an axis at the maximum distance d_{max}. The Robot then approaches the Car and waits for the Car to unlock. Once the Car unlocks, the measurement of the unlock area is marked as successful, and the distance of the Robot from the Car is saved in d^i_{unl}, where i is the number of the axis. Thus, the point at which the Car unlocked is identified by the position of the i-th axis and the corresponding measured distance d^i_{unl}. If the Car fails to unlock before the Robot reaches the minimum distance from the Car d_{min}, the measurement of the unlock area on the i-th axis is marked as unsuccessful.

Once the measurement of the unlock area on the i-th axis is complete, either successfully or unsuccessfully, the Robot retreats from the Car and waits for the Car to lock. Once the Car locks, the measurement of the lock area is marked as successful, and the distance of the Robot from the Car is saved in d^i_{loc}. Similarly to the previous measurement, the point at which the Car locked is thus identified by the position of the i-th axis and the measured distance d^i_{loc}. If the Car fails to lock before the Robot reaches the maximum distance from the Car d_{max}, the measurement of the lock area on the i-th axis is marked as unsuccessful.

When both measurements on the i-th axis are complete, the Robot moves to the next axis. A visualisation of the described flow is in Fig. 2.1.

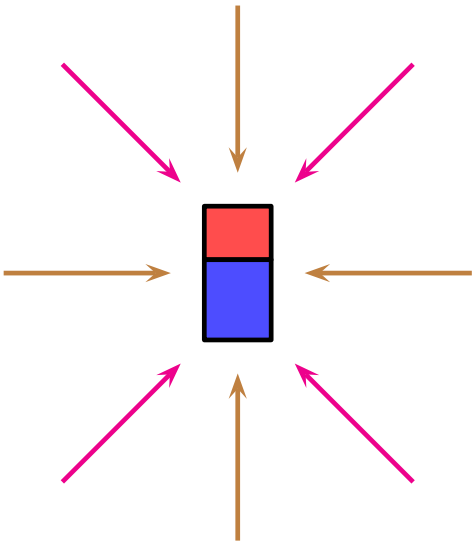
The values $d_{min} \in \mathbb{R}^+_0$, $d_{max} \in (d_{min}, +\infty)$, and $d_{mes} \in (0, +\infty)$ are the parameters of the *Autolock* test scenario.

2.2 Key press scenario

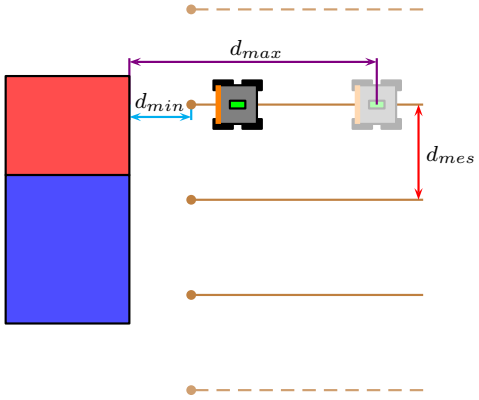
In the *Key press* test scenario, the Robot presses a button on the Remote key while the Operating software expects the Receiver to receive and process the signal. The Robot repeats the button press at various angles around the Car to measure how the angle affects the maximum distance at which the communication between the Remote key and the Receiver operates as intended.

The two-dimensional coordinate system in which the test is conducted is a polar coordinate system. The reference point (i.e., the origin) is identical to the Car's centre. The reference direction matches the front of the Car, and the rotation φ increases in clockwise orientation. The coordinate system is depicted in Fig. 2.2.

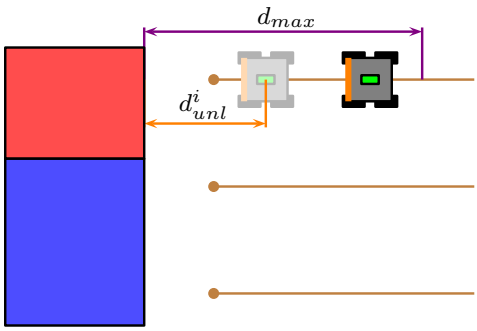
The Robot starts the test at an angle $\varphi_0 = 0$ and a predetermined distance d_{max}, which should be beyond the expected range of the Remote key. The Robot then presses a button on the Remote key. If the Receiver does not register the button press, the communication between the Remote key and the Receiver is considered unreliable, and the Robot approaches



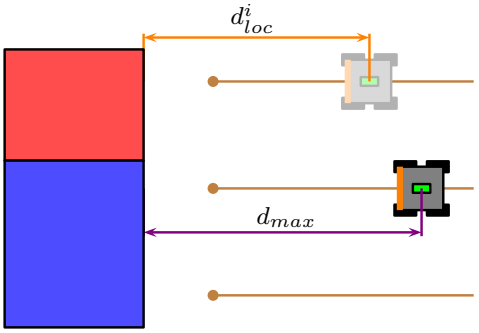
(a) Tested sides and corners of the Car.



(b) The Robot moves toward the Car.



(c) The Robot retreats from the Car.



(d) The Robot moves to the next axis.

Figure 2.1: A visual description of the *Autolock* test scenario. In Fig. 2.1a, the tested sides (brown) and corners (magenta) of the Car are visualised. Fig. 2.1b shows a detailed view of the axes on the right side of the Car. The centre axis is included in the test, as well as all axes distanced $k \cdot d_{mes}$ from the centre axis, for all $k \in \mathbb{N}$ while the axes are within the boundaries of the Car. The test does not include the top- and bottom-most dashed axes, as they exceed the Car's boundaries. The Robot starts the measurement on an axis at the maximum distance d_{max} from the Car. Then, the Robot approaches the Car until either the Car unlocks or the minimum distance d_{min} is reached. Fig. 2.1c visualises a situation in which the Car had unlocked when the Robot (and thus, the Remote key) was distanced d^i_{unl} from the Car, where d^i_{unl} is the measured distance of the unlock area from the Car on the i-th axis. The robot then began to retreat from the car until the car locked or the maximum distance d_{max} was reached. Fig. 2.1d shows that the Car had successfully locked when the Robot was distanced d^i_{loc} from the Car, where d^i_{loc} is the measured distance of the lock area from the Car on the i-th axis. The Robot then moved to the next axis at the maximum distance d_{max}.

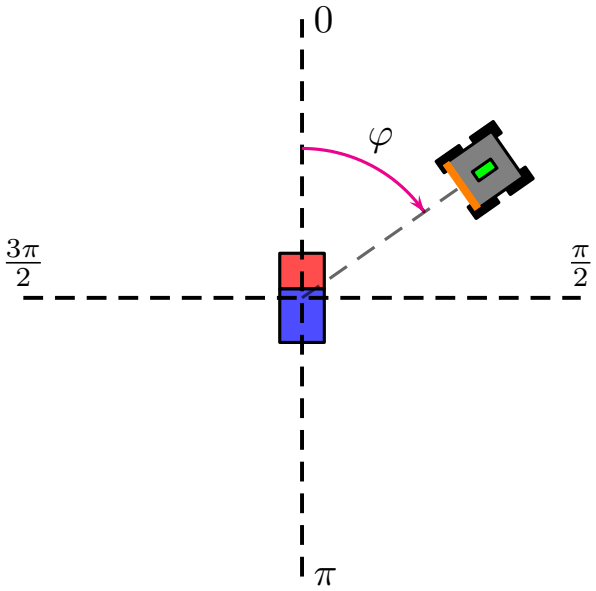
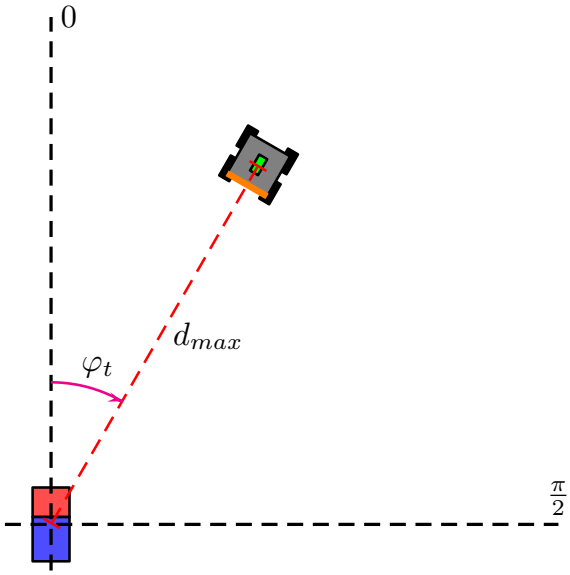


Figure 2.2: Depiction of the coordinate system used in the *Key press* test scenario. The Car is positioned in the centre of the coordinate system, with the front of the Car coloured red and the back of the Car blue. The Robot carries the Remote key, shown in green, and the Robot's front side, in orange, is rotated toward the Car. The angle φ is depicted in magenta.

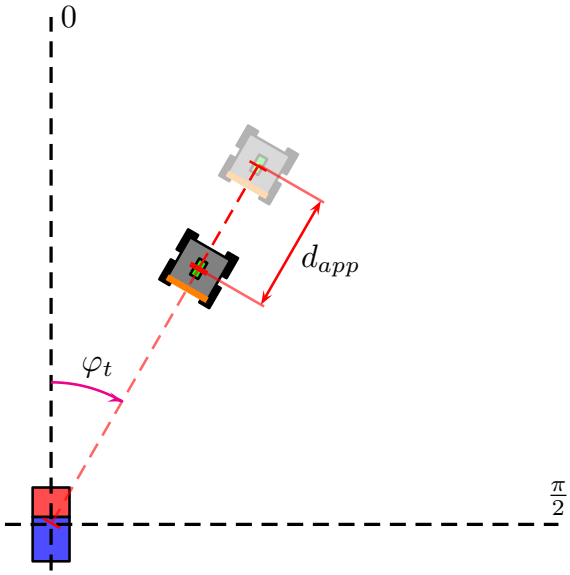
the Receiver by d_{app}. When a button press has been successfully registered, or when a button press at the minimum distance d_{min} has not been successfully registered, the Robot retreats to the predetermined distance d_{max} at the next angle $\varphi_{t+1} = \varphi_t + \varphi_{off}$, as shown in Fig. 2.3.

The Robot continues to perform the test while the condition $\varphi_t < 2\pi$ is satisfied, i.e., until the Robot has circled the Car. Once the condition fails, the test is complete. Therefore, the test ends after the Robot tests the SUT at $\lceil \frac{2\pi}{\varphi_{off}} \rceil$ different angles.

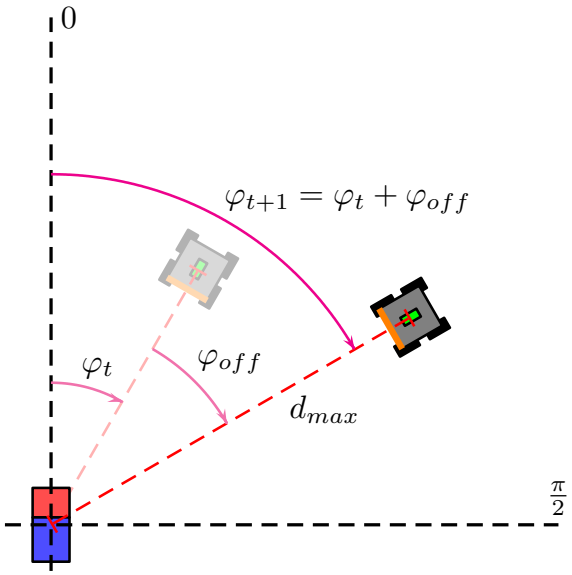
The values $d_{min} \in \mathbb{R}_0^+$, $d_{max} \in [d_{min}, +\infty)$, $d_{app} \in (0, +\infty)$, and $\varphi_{off} \in (0, 2\pi]$ are the parameters of the *Key press* test scenario.



(a) Robot at maximum distance at angle φ_t.



(b) Robot approaches by d_{app} at angle φ_t.



(c) Robot at maximum distance at next angle φ_{t+1}.

Figure 2.3: A depiction of the flow of the *Key press* test scenario. In Fig. 2.3a, the Robot is located at the maximum distance d_{max} at angle φ_t. After an unsuccessful attempt to lock/unlock the Car, the Robot approaches the Car by d_{app} at the same angle φ_t, as shown in Fig. 2.3b. In Fig. 2.3c, after the attempt to lock/unlock the Car has been successful, the Robot retreats to the maximum distance d_{max} at the next angle φ_{t+1}.

Chapter 3

Design

In this chapter, the general design of the proposed solution is discussed. Section 3.1 presents the basic components from which test scenarios can be composed. Section 3.2 discusses the communication between the Operating software and the Robot. Section 3.3 presents the software and hardware architecture. Finally, the results expected to be retrieved from the test are described in Section 3.4.

3.1 Basic actions

The implementation aims to provide a general testing framework that allows for more possible test scenarios than the two described in this thesis. However, the solution is also intended to be easy to use by its users, i.e., the Operator. These two paradigms often contradict each other; in a general solution, the Operator may be required to configure the position of each point the Robot should pass, the velocities the Robot should have at different points, and possibly much more, which could cause several problems. Firstly, the test setup could take the Operator longer than if the Operator performed the test without any automation. Secondly, the Operator could get overwhelmed or confused by the configuration, resulting in an incorrect configuration and, thus, unreliable test results. Lastly, a more qualified Operator could be needed to set up, monitor, and evaluate the test, resulting in additional costs. All of these arguments go against the purposes of test automation, which should lead to faster, more reliable, easily reproducible, and less expensive tests that require fewer human resources.

We designed the solution to provide basic actions/commands that a software developer uses to create a flow suitable for a specific test scenario. The Operator then needs to fill in parameters created by the developer to run the test. The Robot is mostly agnostic on the specific test scenario and only executes commands sent from the Operating software; thus, the software developer mainly needs to alter the Operating software. The only exception is the localisation of the Robot. As is the case for both the scenarios discussed in this thesis, the coordinates communicated between the Operating software and the Robot are relative to the position of the Car. Therefore, the Robot needs to recognise the Car on the map of its surroundings, whereas, in other scenarios, recognition of a different object might be necessary.

The required basic actions are

- moving the Robot to a specific position, called waypoint,
- setting the Robot's speed,
- setting the Robot's direction of movement,
- stopping the Robot's movement,
- moving the Remote key to a specific height,
- and pressing buttons on the Remote key.

A software developer can create and adjust the flow to suit their needs by ordering, repeating, and combining these basic components.

The Robot also informs the Operating software about its current state, i.e., its position, heading, speed, battery level, height of the Remote key, etc. The Operating software can utilise this information in the test's flow.

3.2 Operating software to Robot communication

Communication between the Operating software and the Robot is carried over Wi-Fi, which is a live standard comprising several wireless communication protocols; see [7] and [8]. In today's world, Wi-Fi is widely used by companies and individuals, making its fundamental usage and capabilities well known to the general public. Thus, it is expected that the Operator or other personnel responsible for the test can set up a Wi-Fi network. Additionally, when using Wi-Fi, the range of communication can be virtually limitless, as the Wi-Fi network's coverage can be expanded using range extenders, additional access points, routers, or other devices.

However, in practical terms, the area covered by the Wi-Fi network will always be finite; moreover, the coverage may change in time, e.g., due to noise or access point failures. Therefore, possible communication outages must be taken into account. A simple way to mitigate damage caused by errors in communication is to send the communicated messages repeatedly. To allow for repeated sending, the messages should be idempotent. In our case, this is achieved by sending commands to the Robot containing information about the desired *final* state of the Robot rather than the *change* in the Robot's state; thus, if a message is received multiple times by the Robot, the outcome remains the same. The same principle also applies to messages sent from the Robot to the Operating software.

3.3 Architecture

Both the Operating software and the Robot require an interface to communicate with one another. The Operating software handles that in the Robot (communication) interface, which provides an Application Programming Interface (API) for other software components to communicate with the Robot. The main component that requires services provided by the Robot interface is the Robot controller, which monitors the state of the Robot and sends commands to the Robot according to the currently running test scenario.

The test scenario is selected and parameterised by the Operator in the Operating software's Graphical User Interface (GUI). Moreover, the GUI visualises the progress of the test and informs the Operator about the final results.

In the test scenarios described in this thesis, the Operating software also requires data from the Receiver to register changes in the Car's locks. Traditionally, communication in the automotive industry is carried over Controller Area Network (CAN) bus ([9], [10]). Therefore, the Operating software contains the Car interface, responsible for communication with the Car over the CAN bus.

On the Robot's side, the component responsible for communication with the Operating software is called the Dispatcher node, as it mainly dispatches actions to other components

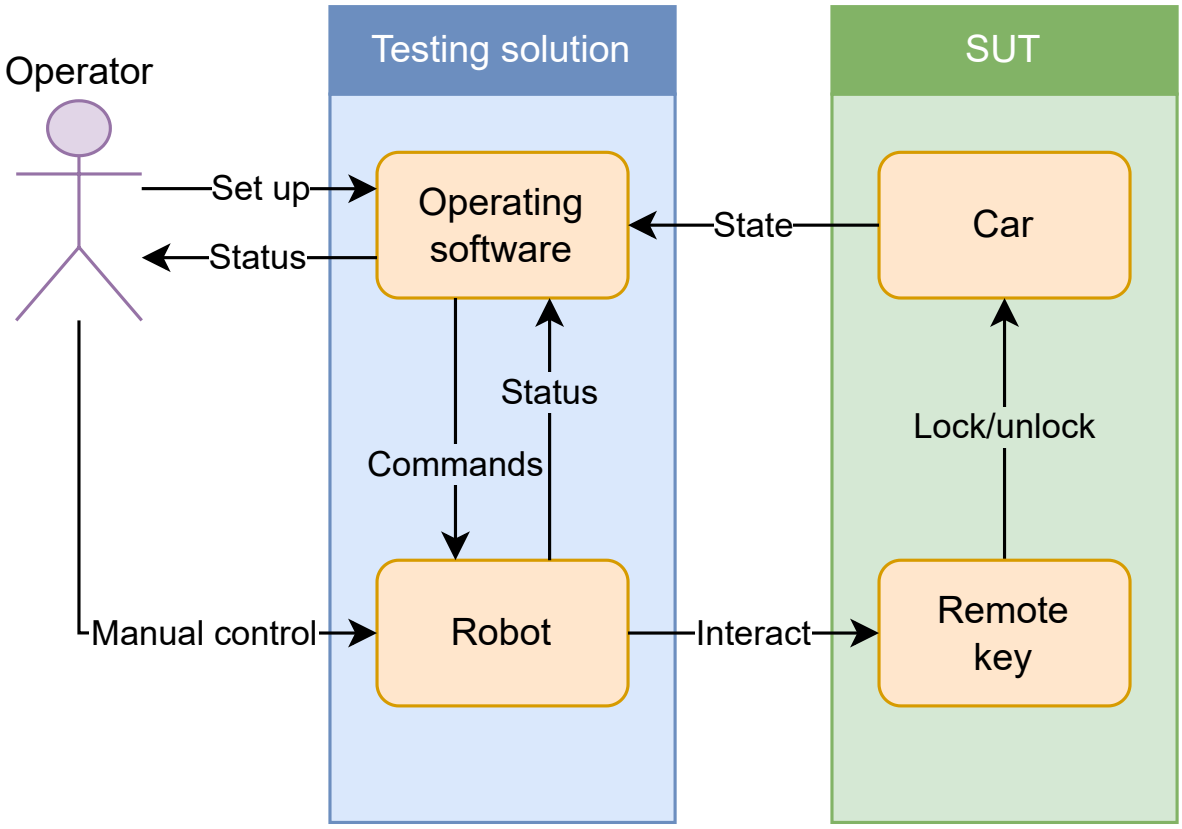


Figure 3.1: Overview of the proposed architecture.

based on commands received from the Operating software. The Dispatcher node is also responsible for transmitting information about the state of the Robot to the Operating software.

The Robot's Peripherals control component handles communication with the peripherals connected to the Robot, comprising the lifting mechanism, the button press mechanism, and several LEDs.

Other components present on the Robot include

- the Drive control, which controls the Robot's wheels,
- Motion planning, which creates a trajectory that the Robot follows to get to the target waypoint,
- Localisation and mapping, which is used to create a map of the Robot's surroundings and find the position of the Robot within this map, allowing navigation and motion planning,
- and Controller input processing, which handles signals from a manual controller that the Operator can use to move the Robot, e.g., to position the Robot before the test or to avoid/prevent a collision.

A high-level visualisation of the proposed architecture can be seen in Fig. 3.1. Detailed views of the Operating software and the Robot are in Fig. 3.2 and Fig. 3.3, respectively. Of the components shown in Fig. 3.2 and Fig. 3.3, this thesis is focused mainly on the Operating software as a whole and the Robot's Dispatcher node.

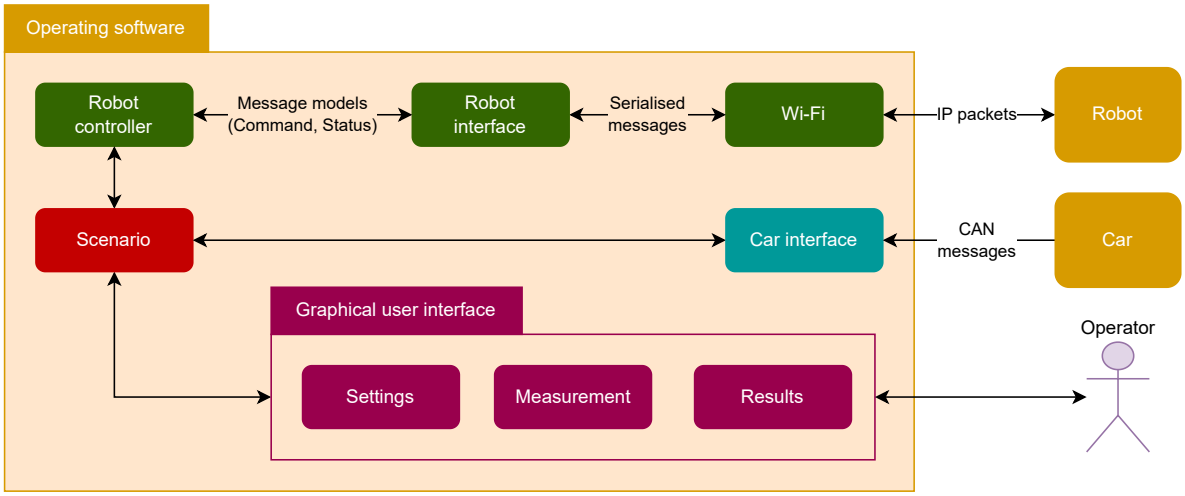


Figure 3.2: A detailed view of the proposed architecture for the Operating software. The labels on some of the connections indicate their primary traffic.

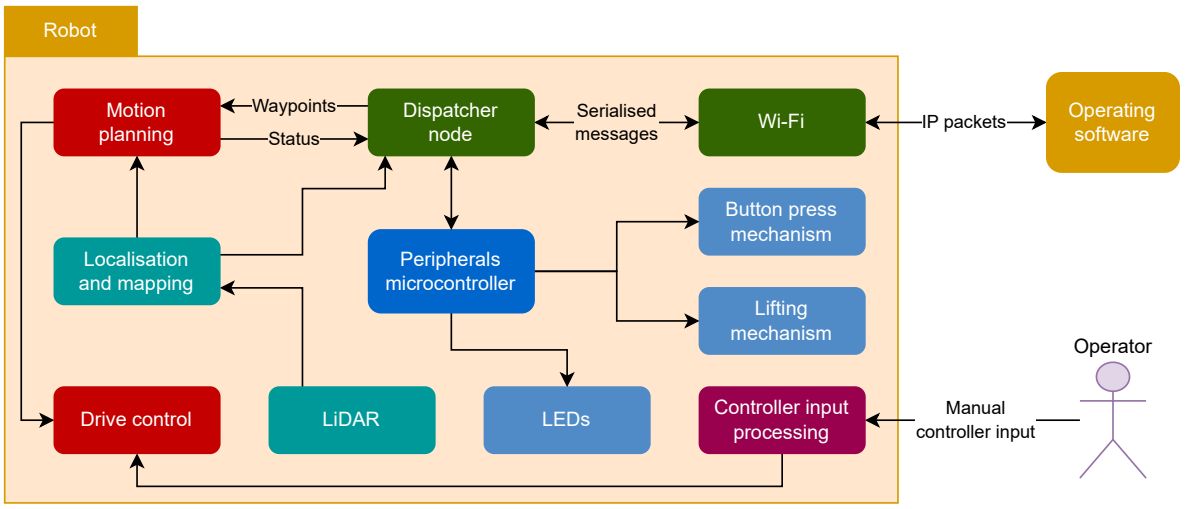


Figure 3.3: A detailed view of the proposed architecture for the Robot. The labels on some of the connections indicate their primary traffic.

3.4 Output

The data obtained during the test are intended to be used by the Operator to verify the SUT. Oftentimes, whether a test has been successful is apparent from a simple visualisation of the test results. It is, therefore, useful for the Operator to be provided with such visualisation by the Operating software.

However, some cases require further analysis, which is impractical to implement directly in the Operating software, as the specifics of the analysis could differ widely between each test run. For this reason, the Operator should also be provided with a detailed output containing the values measured during the test. Such output should be self-explanatory and easy to parse so the Operator can use other tools and software to analyse the test results.

Chapter 4

Implementation

This chapter focuses on the implementation of the proposed system. As designed in Chapter 3, the implementation is divided into two parts. A description of the implementation of the Operating software is in Section 4.1. Section 4.2 follows with the implementation of the Robot's hardware and software.

4.1 Operating software

The Operating software runs on the computer directly connected to the SUT, in our case, the Car's Receiver. Its purpose is to guide the Operator and the Robot through the test scenario. For that, the Operating software provides a GUI where the Operator connects the Operating software to the Robot, configures the test parameters, observes the test progress, and gathers the test results. An introduction of the C# programming language used to develop the Operating software is in Section 4.1.1, with the description of the Operating software itself following in Section 4.1.2. In addition, the Operating software registers whenever the Receiver receives a signal and uses the information to navigate the Robot, further detailed in Section 4.1.3. Further, communication between the Operating software and the Robot is discussed in Section 4.1.4. Lastly, communication of the Operating software with the Car is in Section 4.1.5.

4.1.1 The C# programming language

The Operating software is written in the C# programming language[1] for the .NET platform[2].

C# is a high-level, general-purpose, object-oriented programming language similar to Java[3]. Code written in C# is compiled into the Common Intermediate Language (CIL). The compiled code then runs on a virtual machine called the Common Language Runtime (CLR), which just-in-time compiles the code in the CIL into machine instructions and executes them on the Central Processing Unit (CPU). The CLR also provides services, such as garbage collection, which make development easier for a software developer.

.NET is an implementation of the Common Language Infrastructure (CLI), which comprises the CIL and the CLR. Operating systems supported by .NET[4] include Windows, macOS, several Linux distributions (e.g., Debian, Ubuntu, Fedora, openSUSE, and others), and

[1] https://learn.microsoft.com/en-us/dotnet/csharp/
[2] https://dotnet.microsoft.com/en-us/
[3] https://www.java.com/en/
[4] https://github.com/dotnet/core/blob/main/release-notes/7.0/supported-os.md

popular mobile operating systems Android and iOS. In addition to C#, other languages supported by .NET include F#, a general-purpose functional programming language, C++/CLI, a variant of the C++ programming language modified for the CLI, and Visual Basic, a programming language primarily used for the development of desktop applications targeting the Windows operating system. More information about C# and .NET can be found in [11].

Both C# and .NET have been developed by Microsoft and are currently open-source. The versions used to develop the Operating software are C# 11 and .NET 7, the most recent versions at the beginning of the development.

4.1.2 Graphical user interface

The GUI allows the Operator to connect the Operating software to the Robot, set the test parameters, follow the test progress, and collect the test results. To describe the implementation of the GUI, we first provide an introduction to User Interface (UI) frameworks that can be used with .NET in Section 4.1.2.1. Then, the connection to the Robot is described in Section 4.1.2.2. Lastly, the main window is described in Section 4.1.2.3.

4.1.2.1 UI frameworks for .NET

Nowadays, .NET desktop applications are often built using any of Windows Forms (WinForms), Windows Presentation Foundation (WPF), Multi-platform App UI (MAUI), or Avalonia UI[5] frameworks.

Of these, WinForms is the oldest technology. The interface layout is constructed mainly by drag-and-dropping predefined controls and configuring their appearance. WinForms is currently in maintenance mode, where bug fixes are provided, but new features are not being developed. As the name suggests, Windows is the only operating system supported by WinForms.

WPF has been the most commonly used UI framework for several years, though its development has also stalled recently. Unlike WinForms, Extensible Application Markup Language (XAML) markup is used to define the interface layout, making it easier to version and maintain for larger projects. As with WinForms, WPF only supports Windows.

MAUI is the most recent of the mentioned UI frameworks, having been first released in May 2022. Similarly to WPF, MAUI uses XAML to define the user interface. Having been developed from Xamarin.Forms[6], MAUI supports mobile operating systems iOS and Android as well as desktop operating systems macOS and Windows[7].

Avalonia UI is a UI framework that takes great inspiration from WPF but is still under intense development and supports cross-platform applications. To date, the operating systems supported by Avalonia UI[8] include Windows, macOS, and several Linux distributions (Debian, Ubuntu, Fedora, and possibly others[9]), as well as mobile operating systems iOS and Android. Web applications are also supported using WebAssembly. Of all the mentioned UI frameworks, Avalonia UI is the only one not directly developed by Microsoft; instead, it is a community-driven project developed by individual developers and contributors.

[5] https://avaloniaui.net/
[6] https://dotnet.microsoft.com/en-us/apps/xamarin/xamarin-forms
[7] https://learn.microsoft.com/en-us/dotnet/maui/supported-platforms?view=net-maui-8.0
[8] https://avaloniaui.net/Platforms
[9] https://docs.avaloniaui.net/docs/0.10.x/faq#what-linux-distros-are-supported

For the implementation of the GUI for the Operating software, we decided to use Avalonia UI, version 11, as it provides support for all major desktop operating systems and, due to its similarity to the popular WPF UI framework, an excellent documentation.

4.1.2.2 Connection window

The Operating software connects to the Robot via Transmission Control Protocol (TCP), which provides a reliable bidirectional connection and ensures that messages are received in the correct order and without errors. On the contrary, these features of TCP also cause some problems in implementation. As described in Chapter 3, the Robot periodically sends status information to the Operating software, containing the Robot's position, heading, battery level, and more. Because the communication is wireless and over relatively long distances, the TCP connection can break frequently, causing timeout errors for both the Robot and the Operating software. To seamlessly address the issue without the need to implement additional error handling and connection re-establishment, we decided to use the open-source messaging library ZeroMQ[10], whose publisher-subscriber communication model solved the problem. More information about the ZeroMQ library can be found in [12]. However, to enable the publisher-subscriber pattern, the ZeroMQ library requires two TCP ports to facilitate communication between the Robot and the Operating software.

When the Operating software is launched, a window is shown where the Operator is expected to input the IP address of the Robot and the Operating software's subscriber and publisher ports. The connection window is shown in Fig. 4.1.

To briefly introduce how UI is defined using Avalonia UI, the Avalonia XAML (AXAML) code for the *Connection window* is presented in Listing 4.1. The general structure of the markup should be familiar to anyone who has worked with XAML or Extensible Markup Language (XML).

The first eight lines define a new window, specifying the general properties of the window. Line 5 specifies the width and height of the window, as well as whether the window is resizable, meaning whether the user can change the width and height of the window. Line 6 sets the name of the generated underlying class, using which the window is accessible from both AXAML and C# files. Line 7 specifies the startup location of the window, meaning where the window should initially render on the user's screen. Line 8 sets the window's title, that is, the text at the very top of the window.

Lines 9 through 11 import a style (i.e., visual definitions) from another file. The imported style defines the appearance of the button.

Line 13 contains a definition of a grid. A grid is a collection of visual elements in which each element is located in a particular row and column of the grid, defined by the *Grid.Row* and *Grid.Column* attributes on the contained elements. The values of the attributes *Grid.Row* and *Grid.Column* are indexed from zero. If the attribute is omitted, an implicit value of zero is assumed. The grid defined in line 13 has one (implicit) column and six rows, as written in the *RowDefinitions* attribute of the grid. Each value in the *RowDefinitions* attribute specifies a row's height. The value *Auto* means that the height is inferred from the height of the elements contained in the row, and the value * references the remaining height of the grid, i.e., the total height of the grid minus the sum of the height of all other rows.

[10] https://zeromq.org/

Lines 14 to 27 define the label and the input text box for the IP address, the subscriber port, and the publisher port, respectively. Each element is defined as a grid with three columns and one row.

The label in the first column displays the text specified in the *Content* attribute. The attribute *Target* takes the name of an element that should become focused when the Operator clicks on the label.

The second column is purposely omitted. As the second column of the grid is defined with a width set to *, it will occupy all the remaining space, resulting in the first column being aligned to the left side of the row and the third column to the right side of the row, leaving space in the middle.

The text box in the third column creates an input element in which the Operator can write the desired value. The attribute *Name* specifies the name of the element, which is used by the label in the first column to set focus on the text box when clicked.

The *Text* attribute references the text in the text box. The shown value of the *Text* attribute specifies that the text in the text box is bound to a specific property in the view model. *Mode=TwoWay* means that changes written to the text box by the Operator are stored in the bound property and, conversely, that changes to the bound property are written in the text box. The reason for the second way, from the bound property to the text box, is that after the connection to the Robot has been successfully established, the Operating software saves the IP address and TCP ports to a file, from where the values are retrieved the next time the Operating software is launched. The values are then used to pre-fill the text boxes in the *Connection window*.

The text blocks in lines 29 and 30 define the validation messages shown when the attempt to connect to the robot fails. The messages' *IsVisible* attributes are bound to properties on the view model, making the messages visible only when needed. As the modes of the bindings of the *IsVisible* attributes are unspecified, an implicit *Mode=OneWay* is assumed, causing the element to react to changes in the view model, but not the other way around.

The first message is shown when the values entered by the Operator are not in a valid format, e.g., when the TCP ports are outside the range $0 - 65535$.

The second message is shown when an attempt to connect with the Robot fails. As described earlier in this thesis, the ZeroMQ library is used to communicate between the Operating software and the Robot, making it easier to handle errors when the communication is broken. However, such behaviour of the ZeroMQ library is undesirable when establishing the connection because the Operator could have specified incorrect connection parameters, which the library would hide from the Operator, making it difficult to spot errors. Therefore, before establishing the connection using ZeroMQ, an attempt is made to establish the connection over standard TCP, resulting in an error if the Robot is unavailable. The error is then handled by the *Connection window*, and the second error message is shown to the Operator. The code to establish the connection is in Listing 4.2.

Line 32 in the *Connection window*'s code in Listing 4.1 defines the *Connect button*, whose *Command* attribute specifies the method called when the button is clicked.

Although the top-level grid of the *Connection window* has been defined as having six rows, only five have been specified, with the fourth row missing. Similarly to the design of the columns of the inner grid described earlier, the fourth row with a height set to * creates a space between the input text boxes and the *Connect button* with the validation messages.

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        Width="300" Height="180" CanResize="False"
        x:Class="Prazak.Diploma.Gui.Views.ConnectionWindow"
        WindowStartupLocation="CenterScreen"
        Title="Connection">
  <Window.Styles>
    <StyleInclude Source="/Views/Styles/CommonStyles.axaml"/>
  </Window.Styles>

  <Grid RowDefinitions="Auto,Auto, Auto,*, Auto,Auto" Margin="10">
    <Grid ColumnDefinitions="Auto,*,Auto" Margin="0 0 0 4">
      <Label Content="IP address:" Target="IpAddressTextBox"/>
      <TextBox Name="IpAddressTextBox" Grid.Column="2" Width="150" Text="{Binding IpAddress,
          Mode=TwoWay}"/>
    </Grid>

    <Grid Grid.Row="1" ColumnDefinitions="Auto,*,Auto" Margin="0 0 0 4">
      <Label Content="Subscriber port:" Target="SubscriberPortTextBox"/>
      <TextBox Name="SubscriberPortTextBox" Grid.Column="2" Width="150" Text="{Binding
          SubscriberPort, Mode=TwoWay}"/>
    </Grid>

    <Grid Grid.Row="2" ColumnDefinitions="Auto,*,Auto">
      <Label Content="Publisher port:" Target="PublisherPortTextBox"/>
      <TextBox Name="PublisherPortTextBox" Grid.Column="2" Width="150" Text="{Binding
          PublisherPort, Mode=TwoWay}"/>
    </Grid>

    <TextBlock Grid.Row="4" Text="The IP address or port provided is invalid." Margin="0 0 0 2"
     TextAlignment="Left" Foreground="Red" IsVisible="{Binding ShowFormatInvalidMessage}"/>
    <TextBlock Grid.Row="4" Text="Connection failed, check the parameters." Margin="0 0 0 2"
        TextAlignment="Left" Foreground="Red" IsVisible="{Binding
        ShowConnectionFailedMessage}"/>

    <Button Name="ConnectButton" Command="{Binding ConnectButtonClickCommand}" Width="280"
        Grid.Row="5" HorizontalAlignment="Center" Content="Connect" Classes="BigButton"/>
  </Grid>
</Window>
```

Listing 4.1: AXAML code defining the *Connection window*.

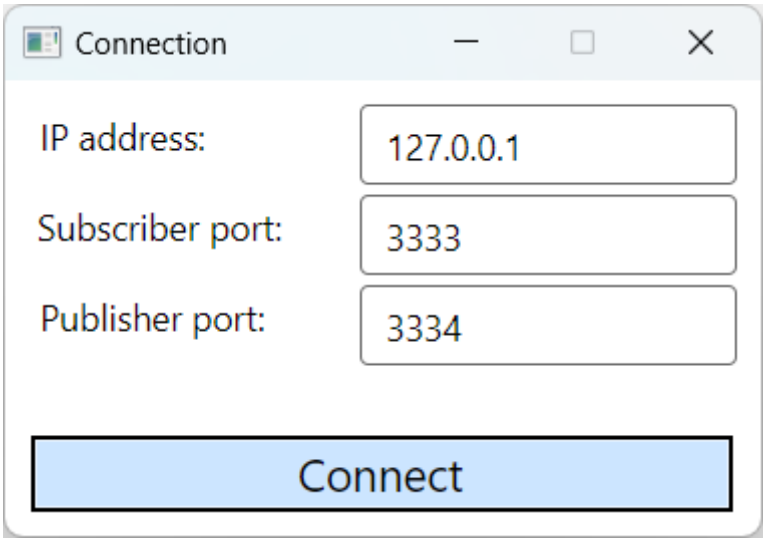


Figure 4.1: A screenshot of the connection window of the Operating software. The Subscriber and Publisher ports are named from the Operating software's point of view, whereas the Internet Protocol (IP) address is of the Robot.

```
1     public void Connect(string ipAddress, int subscriberPort, int publisherPort)
2     {
3         using (var tcpClient = new TcpClient())
4         {
5             var task = Task.Run(() => tcpClient.Connect(IPAddress.Parse(ipAddress),
                  subscriberPort));
6             if (!task.Wait(_configuration.ConnectionAttemptTimeout))
7                 throw new TimeoutException($"Connection attempt to {ipAddress}:{subscriberPort}
                      timed out after {_configuration.ConnectionAttemptTimeout} ms.");
8         }
9
10        using (var tcpClient = new TcpClient())
11        {
12            var task = Task.Run(() => tcpClient.Connect(IPAddress.Parse(ipAddress),
                  publisherPort));
13            if (!task.Wait(_configuration.ConnectionAttemptTimeout))
14                throw new TimeoutException($"Connection attempt to {ipAddress}:{publisherPort}
                      timed out after {_configuration.ConnectionAttemptTimeout} ms.");
15        }
16
17        _zmqSubscriber.Connect($"tcp://{ipAddress}:{subscriberPort}");
18        _zmqSubscriber.SubscribeToAnyTopic();
19
20        _zmqPublisher.Connect($"tcp://{ipAddress}:{publisherPort}");
21
22        _ = HandleReceivedMessages();
23
24        IsConnected = true;
25    }
```

Listing 4.2: C# code handling the connection attempt to the Robot.

```
1  <Window xmlns="https://github.com/avaloniaui"
2          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
3          xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
4          xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
5          xmlns:controls="using:Prazak.Diploma.Gui.Views"
6          MinWidth="900" MinHeight="550" Width="900" Height="550"
7          x:Class="Prazak.Diploma.Gui.Views.MainWindow"
8          Title="Application">
9    <Grid RowDefinitions="Auto, *">
10     <controls:HeaderBar BorderBrush="Black" BorderThickness="0 0 0 2"/>
11     <controls:MainScreen Grid.Row="1"/>
12   </Grid>
13 </Window>
```

Listing 4.3: AXAML code defining the *Main window*.

4.1.2.3 Main window

Once the connection to the robot has been established, the *Main window* is shown. The *Main window* consists of a *Header bar* at the top and the *Main screen* filling the rest of the window, as can be seen in Listing 4.3. The fifth line in the Listing 4.3 allows using elements from the namespace *Prazak.Diploma.Gui.Views* using the prefix *controls*, as shown in lines 10 and 11.

Header bar

The *Header bar*, see Fig. 4.2, visualises the progress through the test in terms of individual screens, which follow in the sequence of the *Welcome screen*, the *Measurement type selection screen*, the *Measurement screen*, and the *Protocol screen*.

Furthermore, a battery icon showing the status of the Robot's battery is present, as well as a button to open the settings window, described later in the text.

Additionally, the header bar contains a warning triangle, which is only visible when a notification is received from the Robot. Upon mouse hover, the triangle displays the message received from the robot. A code snippet of the definition of the warning triangle is given in Listing 4.4.

The style in lines 2 to 15 defines an animation played when the *changed* class is added to the *WarningSymbol* element. The animation linearly increases the element's scale from 1.0 to 1.2 over an interval of 200 ms, which should attract the Operator's attention. The *changed* class is added to the *WarningSymbol* element from the code-behind whenever a notification is received from the Robot. The notification text is added to the *ToolTips* stack panel, in lines 18 to 21, making it visible when the Operator hovers with the mouse over the *WarningSymbol* element.

The triangular shape of the element is achieved using the *Path* element, which accepts a mini-language[11] similar to Scalable Vector Graphics (SVG)[12] in the *Data* attribute defining its shape.

[11] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/graphics-multimedia/path-markup-syntax
[12] https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Paths

```
<UserControl.Styles>
  <Style Selector="controls|WarningSymbol.changed">
    <Style.Animations>
      <Animation Duration="0:0:0.200">
        <KeyFrame Cue="0%">
          <Setter Property="ScaleTransform.ScaleX" Value="1"/>
          <Setter Property="ScaleTransform.ScaleY" Value="1"/>
        </KeyFrame>
        <KeyFrame Cue="100%">
          <Setter Property="ScaleTransform.ScaleX" Value="1.2"/>
          <Setter Property="ScaleTransform.ScaleY" Value="1.2"/>
        </KeyFrame>
      </Animation>
    </Style.Animations>
  </Style>
</UserControl.Styles>
<Panel Background="Transparent" >
  <ToolTip.Tip>
    <StackPanel Name="ToolTips">
    </StackPanel>
  </ToolTip.Tip>
  <Path Data="M 25 6 L 47 44 L 3 44 L 25 6"
        Stroke="#CC0000" StrokeThickness="1"
        StrokeLineCap="Round" StrokeJoin="Round"
        Fill="#CC0000"/>
  <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
             Margin="0 8 0 0" FontFamily="Helvetica"
             Foreground="#E6E6E6" FontSize="22">
    !
  </TextBlock>
</Panel>
```

Listing 4.4: AXAML code snippet defining the warning triangle.

Figure 4.2: A screenshot of the *Header bar*.

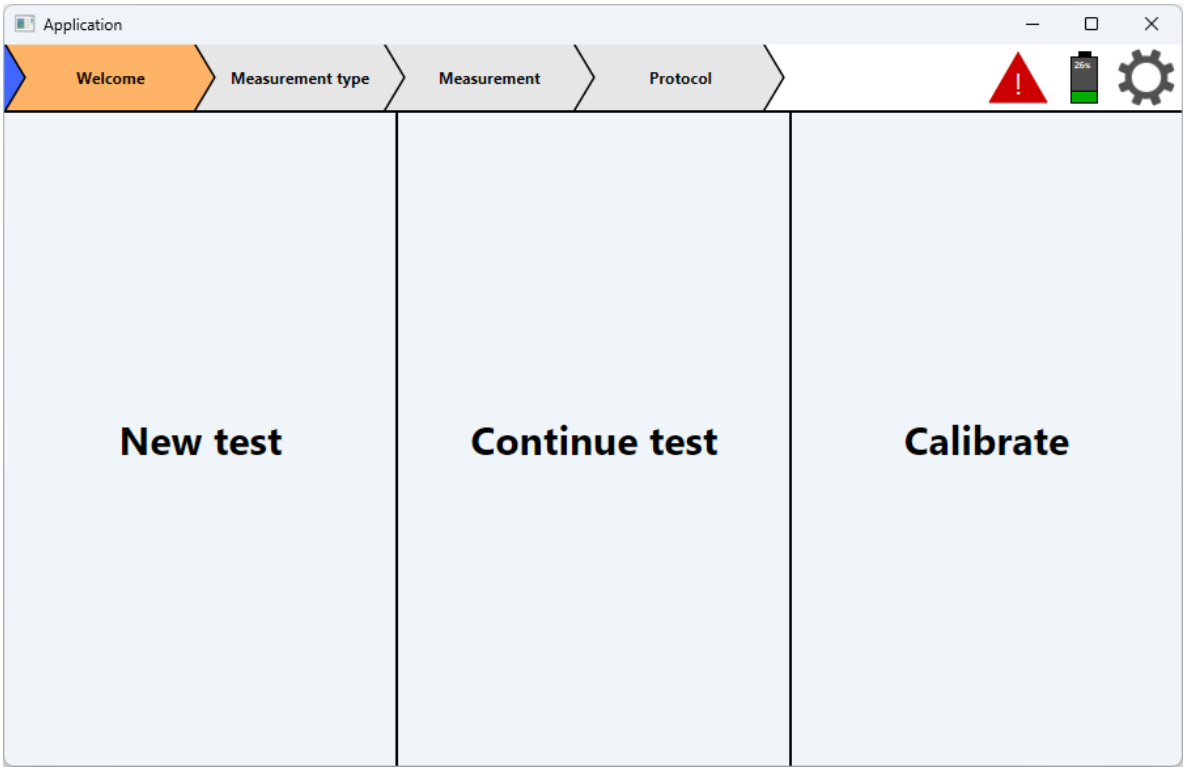


Figure 4.3: A screenshot of the *Main window* showing the *Welcome screen*.

The code for the *Header bar* is in Listing 4.5. The *Classes* attributes attached to the *HeaderItem* elements create a logical group of elements with some common properties. On its own, the attribute does not change the appearance of an element but can be selected by a style selector, similarly to the Cascading Style Sheets (CSS) language[13], which then specifies a style for all elements within the class.

The syntax used for the *Classes.Selected* and *Classes.Finished* attributes specifies a conditional class which binds to a Boolean variable. When the Boolean variable resolves to *true*, the class (in our case, *Selected* or *Finished*) is added to the elements *Classes* attribute. Otherwise, the class is absent in the *Classes* attribute, allowing for dynamic changes in an element's appearance.

Main screen

The *Main screen* of the *Main window* contains all the screens used throughout the test, always showing only the currently active screen. A code snippet defining the *Main screen* is provided in Listing 4.6.

Welcome screen

Initially, the Operating software shows the *Welcome screen*, in Fig. 4.3. The *Welcome screen* contains three buttons, which the Operator can click to select the desired action.

[13] `https://developer.mozilla.org/en-US/docs/Web/CSS`

```
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:controls="using:Prazak.Diploma.Gui.Views"
             mc:Ignorable="d" d:DesignWidth="900" d:DesignHeight="50"
             x:Class="Prazak.Diploma.Gui.Views.HeaderBar">
  <UserControl.Styles>
    <StyleInclude Source="/Views/Styles/HeaderStyles.axaml"/>
  </UserControl.Styles>

  <Grid ColumnDefinitions="Auto,*,Auto" Name="ProgressionGrid">
    <StackPanel Orientation="Horizontal">
      <controls:HeaderItemArrow Background="{DynamicResource DarkBlue}"/>

      <controls:HeaderItem
          Text="Welcome" Classes="Selected" ZIndex="-1"
          Classes.Selected="{Binding WelcomeScreenSelected}" Classes.Finished="{Binding
              WelcomeScreenFinished}"/>

      <controls:HeaderItem
        Text="Measurement type" ZIndex="-2"
        Classes.Selected="{Binding MeasurementSelectionScreenSelected}"
            Classes.Finished="{Binding MeasurementSelectionScreenFinished}"/>

      <controls:HeaderItem Text="Measurement" Margin="-15 0 0 0" ZIndex="-3"
        Classes.Selected="{Binding MeasurementScreenSelected}" Classes.Finished="{Binding
            MeasurementScreenFinished}"/>

      <controls:HeaderItem Text="Protocol" Margin="-15 0 0 0" ZIndex="-4"
         Classes.Selected="{Binding ProtocolScreenSelected}"/>
    </StackPanel>

    <StackPanel Grid.Column="2" Orientation="Horizontal" Spacing="15" Margin="5 0">
      <controls:WarningSymbol/>
      <controls:BatteryIcon/>
      <controls:SettingsButton/>
    </StackPanel>
  </Grid>
</UserControl>
```

Listing 4.5: AXAML code defining the *Header bar*.

```
<Grid>
  <controls:WelcomeScreen IsVisible="{Binding WelcomeScreenSelected}"/>
  <controls:MeasurementSettingsScreen IsVisible="{Binding SettingsScreenSelected}"/>
  <controls:MeasurementScreen IsVisible="{Binding MeasurementScreenSelected}"/>
  <controls:ProtocolScreen IsVisible="{Binding ProtocolScreenSelected}"/>
</Grid>
```

Listing 4.6: AXAML code snippet defining the *Main screen*.

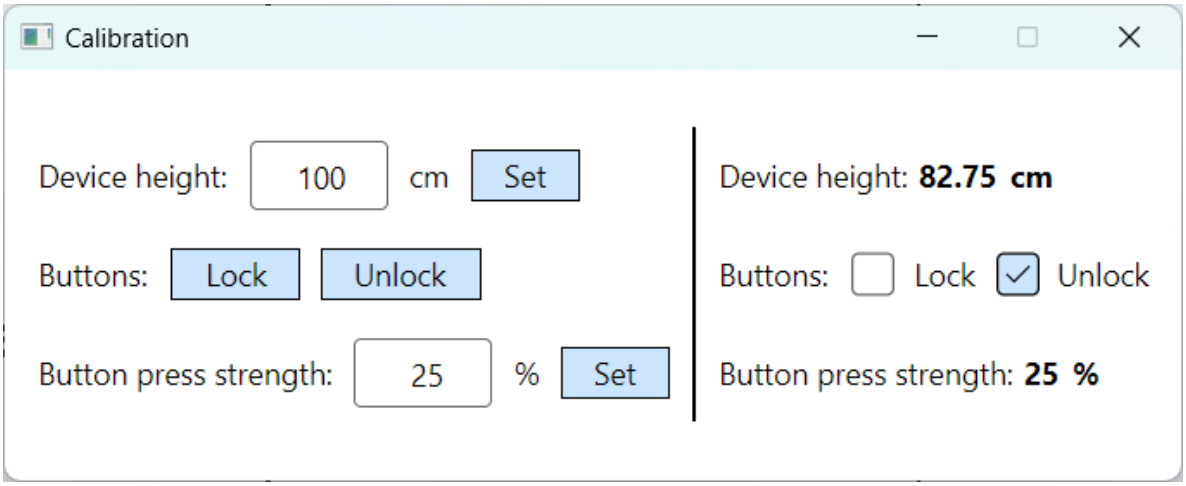


Figure 4.4: A screenshot of the *Calibration window*.

Clicking the *New test* button transitions the Operator to the *Measurement type selection screen* to choose the desired test scenario, that is, either the *Autolock* or the *Key press* scenario. The *Measurement type selection screen* is discussed later in the text.

The *Continue test* button opens a file picker where the Operator can select a previously saved JSON[14] file. The Operating software reads the file to reconstruct a past state of the Operating software so that the test saved in the file can be resumed. It contains information about the parameters used to run the test, results measured during the test, and additional values needed to recreate the internal state of the Operating software correctly. If a valid file is selected and the state is restored successfully, the Operating software switches to the *Measurement screen*, discussed later, where the Operator can continue the loaded test.

Calibration window

The *Calibrate* button opens the *Calibration window* in which the correct functioning of the peripherals connected to the Robot can be verified. The *Calibration window* shows the Operator the most up-to-date values the Operating software has received from the Robot regarding the height of the lifting mechanism and the buttons pressed on the Remote key. The button press strength, which determines the force used to press a button on the Remote key, is also indicated. Additionally, the Operator can input values specifying the desired height of the lifting mechanism, the buttons to be pressed on the Remote key, the force to be used to press a button, and send the information from the *Calibration window* to the Robot to observe if the Robot and the peripherals react correctly. A screenshot of the *Calibration window* is available in Fig. 4.4.

Measurement type selection screen

The *New* button on the *Welcome screen* opens the *Measurement type selection screen*, shown in Fig. 4.5. The *Measurement type selection screen* is visually similar to the *Welcome screen* and offers the Operator the option to select the desired test scenario, either *Autolock* or *Key press*. The general flow of both test scenarios from the Operator's point of view is

[14] `https://www.json.org/`

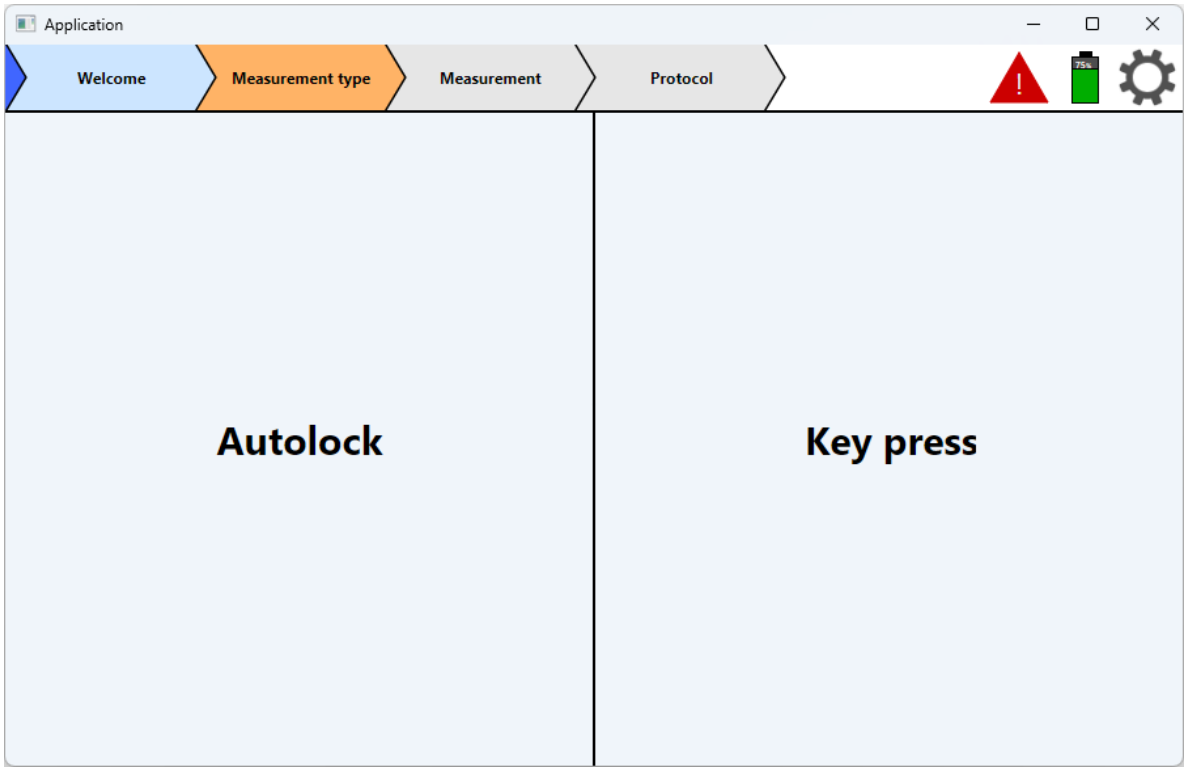


Figure 4.5: A screenshot of the *Main window* showing the *Measurement type selection screen*.

the same; however, each scenario requires different test parameters, is composed of different commands sent to the Robot, and expects different outputs from the Receiver.

Measurement settings screen (Autolock scenario)

Clicking the *Autolock* button on the *Measurement type selection screen* shows the *Measurement settings screen* specific for the *Autolock* test scenario, see Fig. 4.6. There, the Operator sets the parameters of the test.

In the *Autolock* test scenario, the Robot carries the Remote key toward the Car until the Car becomes unlocked, after which the Robot starts retreating from the Car until the Car locks itself again. This measurement is performed with the Robot moving toward each corner of the Car and toward each side of the Car.

The Robot starts each measurement at the *Starting distance* from the Car. Then, the Robot moves by the *Straightening distance* toward the Car to allow for heading correction. Afterwards, the Robot is commanded to move toward a position with the *Minimum distance* from the Car. Once the Operating software registers that the Car has been unlocked, the Robot is commanded to move away from the Car toward the position whose distance corresponds to the *Starting distance* minus the *Straightening distance*. When the Car locks itself, the Robot moves to the next axis, and the measurement is repeated.

The axes are calculated from the *Measurements distance*. The corners of the Car are always included, as well as the centres of all sides (i.e., left, front, right, rear). Then, the axes located $k \cdot$ *Starting distance* from the centre of each side are included, where k is a positive integer that increases as long as the axis is located within the boundaries of the Car.

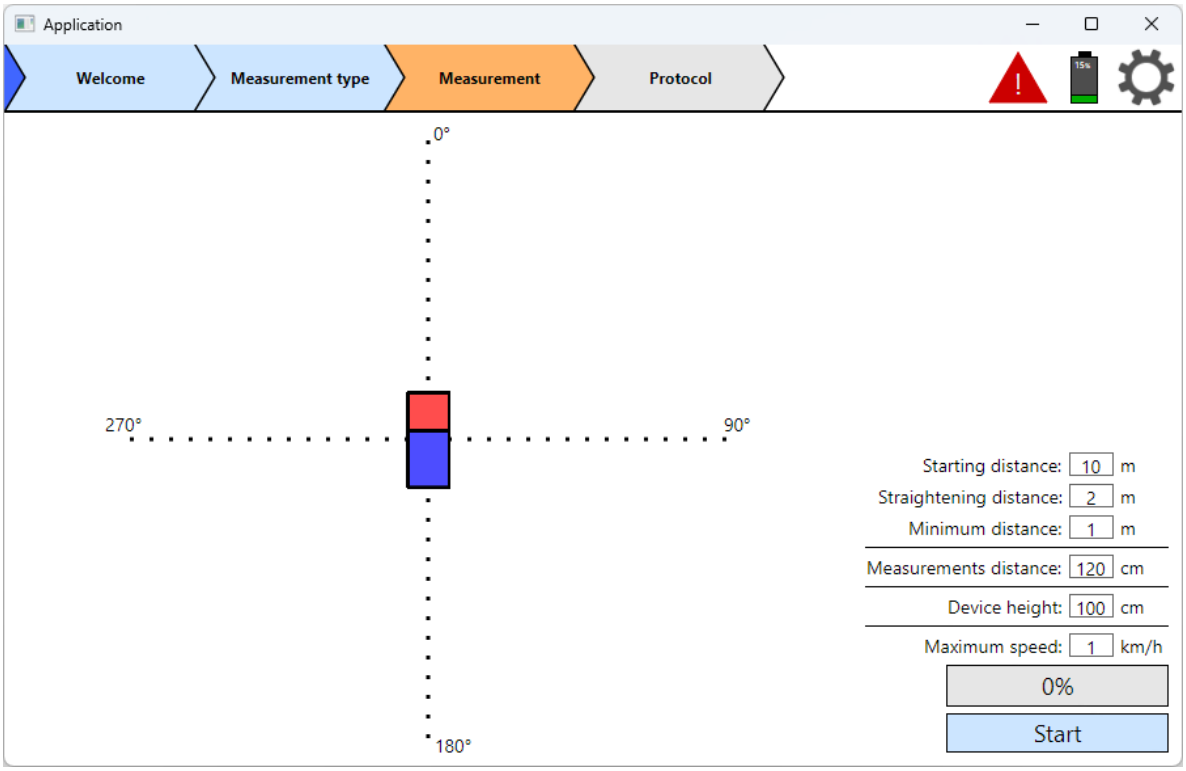


Figure 4.6: A screenshot of the *Main window* showing the *Measurement settings screen* specific to the *Autolock* test scenario.

The *Device height*, in centimetres, specifies the height at which the Robot should carry the Remote key. The *Maximum speed*, in kilometres per hour, sets the maximum speed at which the Robot is allowed to travel, which the Operator can use to adjust the driving properties of the Robot for different environments, e.g., a lower speed might be desirable on an uneven surface to prevent possible damage to the Robot's hardware.

Measurement screen (Autolock scenario)

Pressing the *Start* button initiates the test, and the *Measurement screen* specific to the *Autolock* scenario is displayed, as shown in Fig. 4.7. There, the Operator can see the current position of the Robot relative to the Car, monitor the progress of the test, temporarily pause the test, and save the test's progress, to be loaded later using the *Continue* button on the *Welcome screen* mentioned above. The progress of the test, displayed as a percentage, indicates how many radials from the total number of radials have been completed.

Protocol screen (Autolock scenario)

When all radials have been completed, the Operating software transitions to the *Protocol screen*, shown in Fig. 4.8. The *Protocol screen* allows the Operator to save the results as a Comma-Separated Values (CSV) text file and a Portable Network Graphics (PNG) image. An image generated from the *Protocol screen* can be seen in Fig. 4.9. The generated image is almost identical to the *Protocol screen* since it reuses the same components but renders them

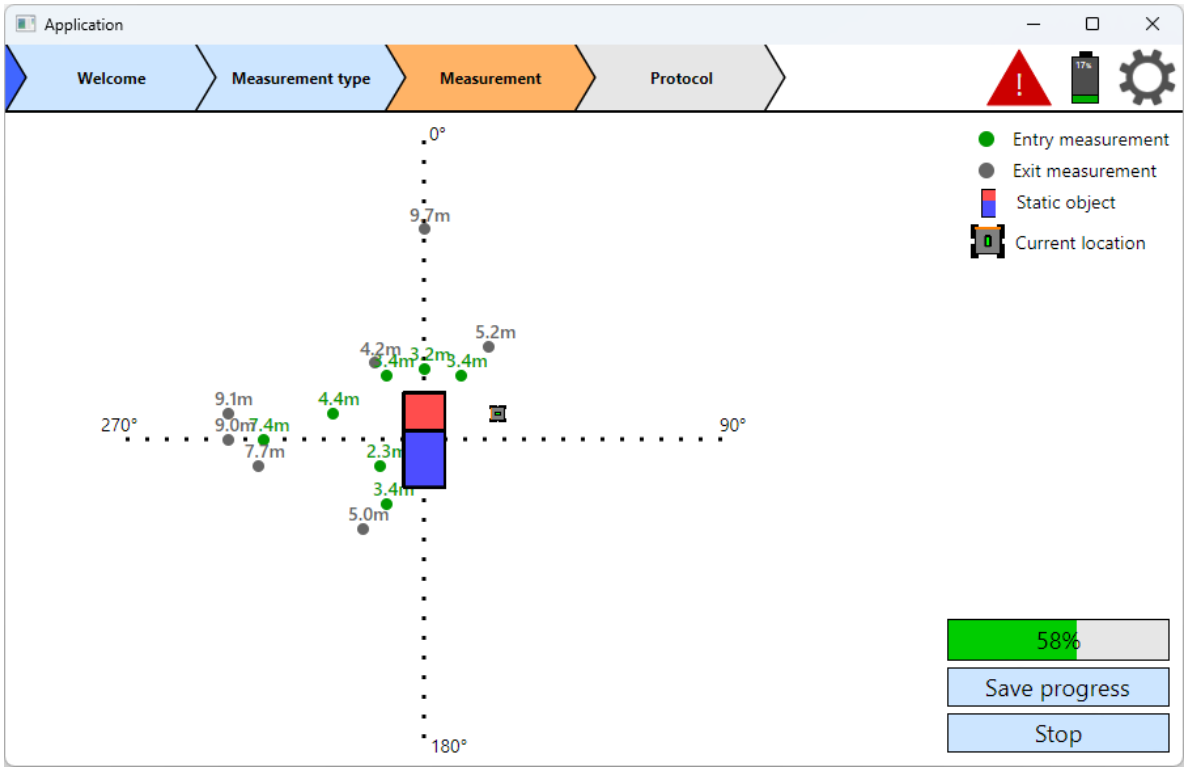


Figure 4.7: A screenshot of the *Main window* showing the *Measurement screen* specific to the *Autolock* test scenario.

to a bitmap rather than a screen, which is then saved to a file. The location of the generated files is specified in the *Settings window*, which is described later in the text.

Measurement settings screen (Key press scenario)

On the *Measurement type selection screen*, in Fig. 4.5, the *Key press* button opens the *Measurement settings screen* specific to the *Key press* test scenario, shown in Fig. 4.10. As in the *Autolock* scenario, the Operator uses the *Measurement settings screen* to set the test parameters.

The *Key press* test scenario is designed so that the Robot starts in front of the Car, at the axis labelled 0°. The distance, in metres, at which the Robot starts the test is called the *Starting distance*. Then, the Robot moves a few meters toward the Car to fix its heading. This distance is called the *Straightening distance*. After that, the Robot is prepared to begin the test. A button is pressed on the Remote key, and the result is read from the Receiver by the Operating software.

On success, the Robot continues to the next radial by adding the *Degree between radials* to the current angle between the Robot and the 0° radial and retreating to the *Starting distance*.

If the attempt is unsuccessful, the Robot approaches the Car by the *Approach distance* and presses the Remote key button again. This behaviour is repeated until either the attempt is successfully registered by the Operating software (and thus, the Receiver) or the *Minimum distance* is reached. In both cases, the Robot continues to the next radial, as described above.

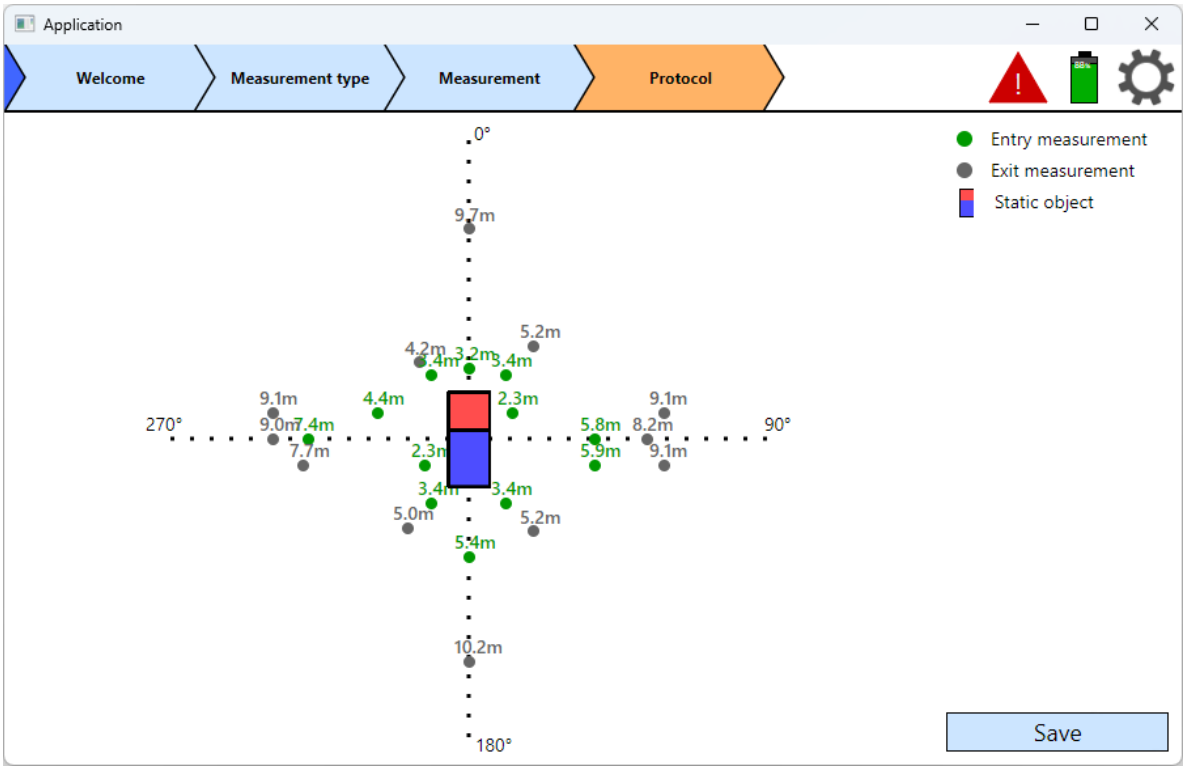


Figure 4.8: A screenshot of the *Main window* showing the *Protocol screen* specific to the *Autolock* test scenario.

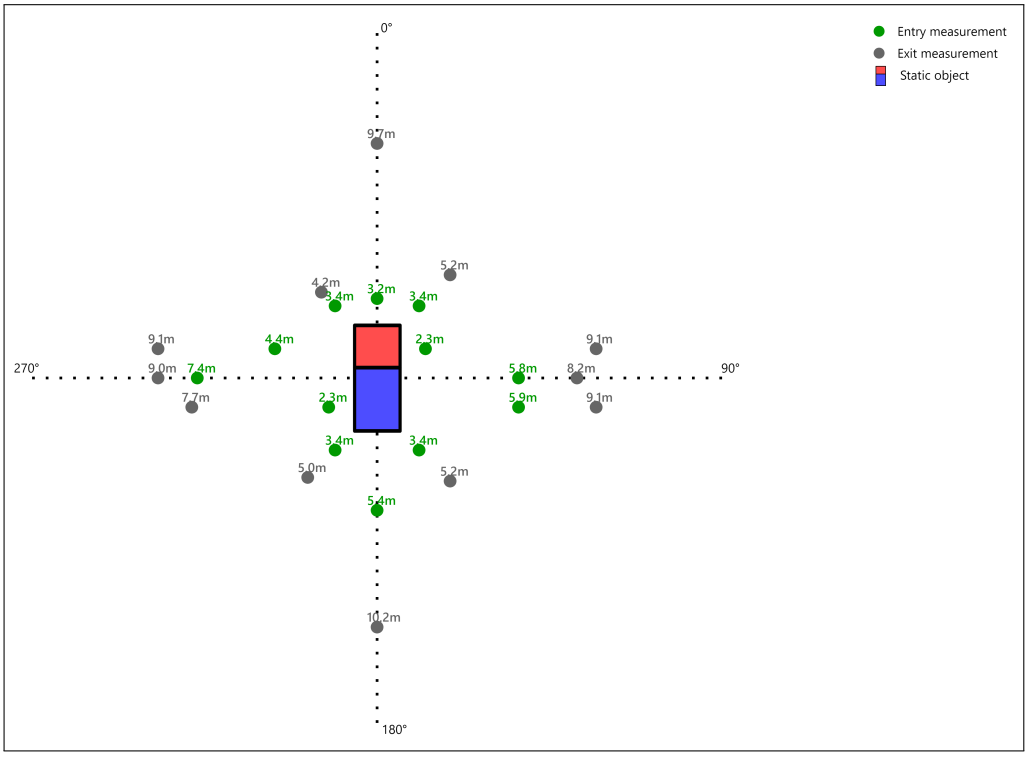


Figure 4.9: An image generated by the Operating software from the *Protocol screen* specific to the *Autolock* scenario showing measurement results.

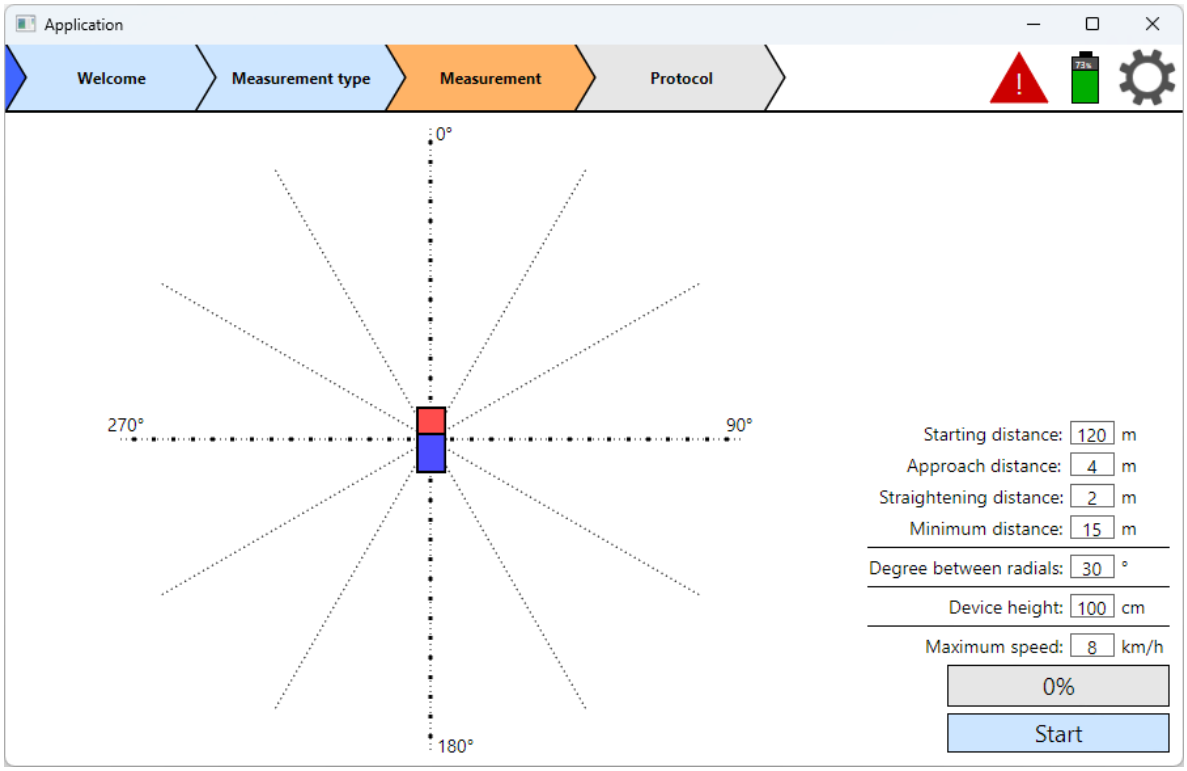


Figure 4.10: A screenshot of the *Main window* showing the *Measurement settings screen* specific to the *Key press* test scenario.

The parameter *Device height*, in centimetres, specifies the height at which the Robot should hold the Remote key. The *Maximum speed*, in kilometres per hour, sets the maximum speed at which the Robot is allowed to travel.

Measurement screen (Key press scenario)

Pressing the *Start* button initiates the test, and the *Measurement screen* specific to the *Key press* scenario is displayed, as shown in Fig. 4.11. The controls available to the Operator remain the same as in the *Autolock* test scenario with only minor visual adjustments, most notably, showing the axis lines. The axis lines were not present in the *Autolock* scenario as their position depends on the car's dimensions, which may be adjusted by the Robot during the test, possibly confusing the Operator.

Protocol screen (Key press scenario)

When the test is completed, the Operating software transitions to the *Protocol screen*, shown in Fig. 4.12, where the final results of the test run can be viewed. Similarly to the *Autolock* test scenario, the *Save* button allows the Operator to save the test results. The test results are saved as a CSV text file and a PNG image. The location of the generated files is specified in the *Settings window*. An image generated from the *Protocol screen* can be seen in Fig. 4.13.

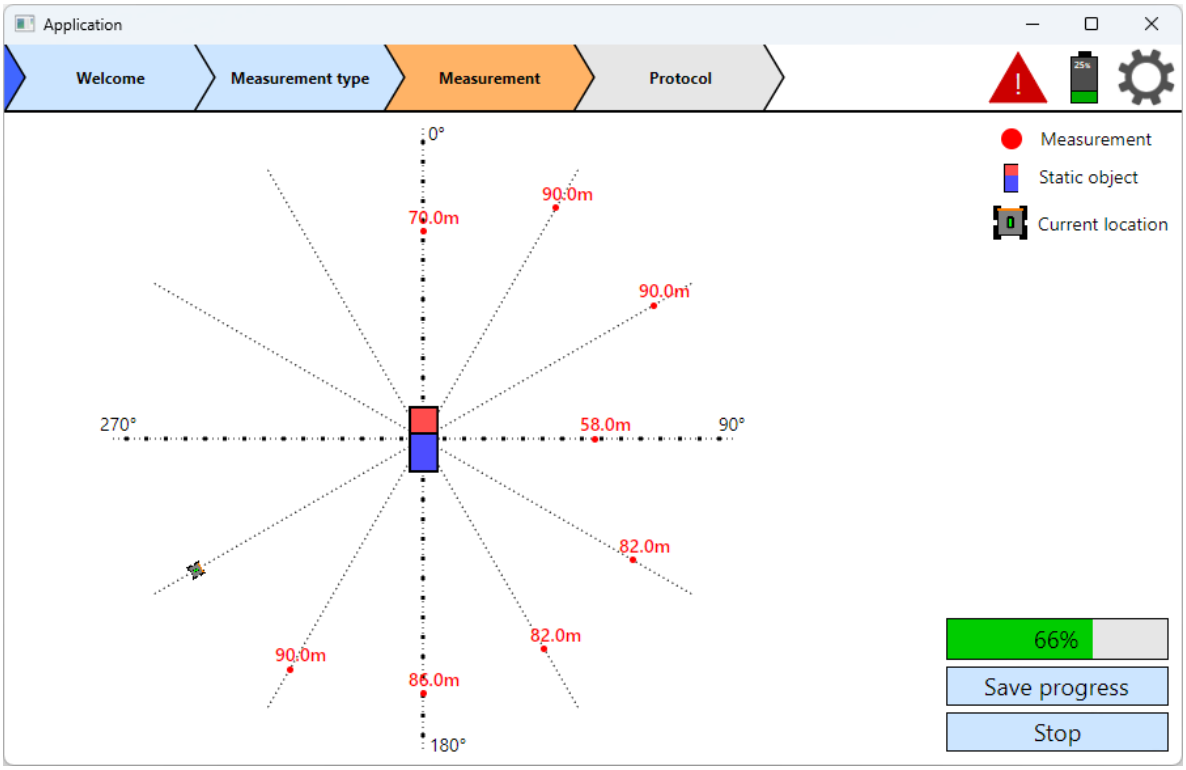


Figure 4.11: A screenshot of the *Main window* showing the *Measurement screen* specific to the *Key press* test scenario.

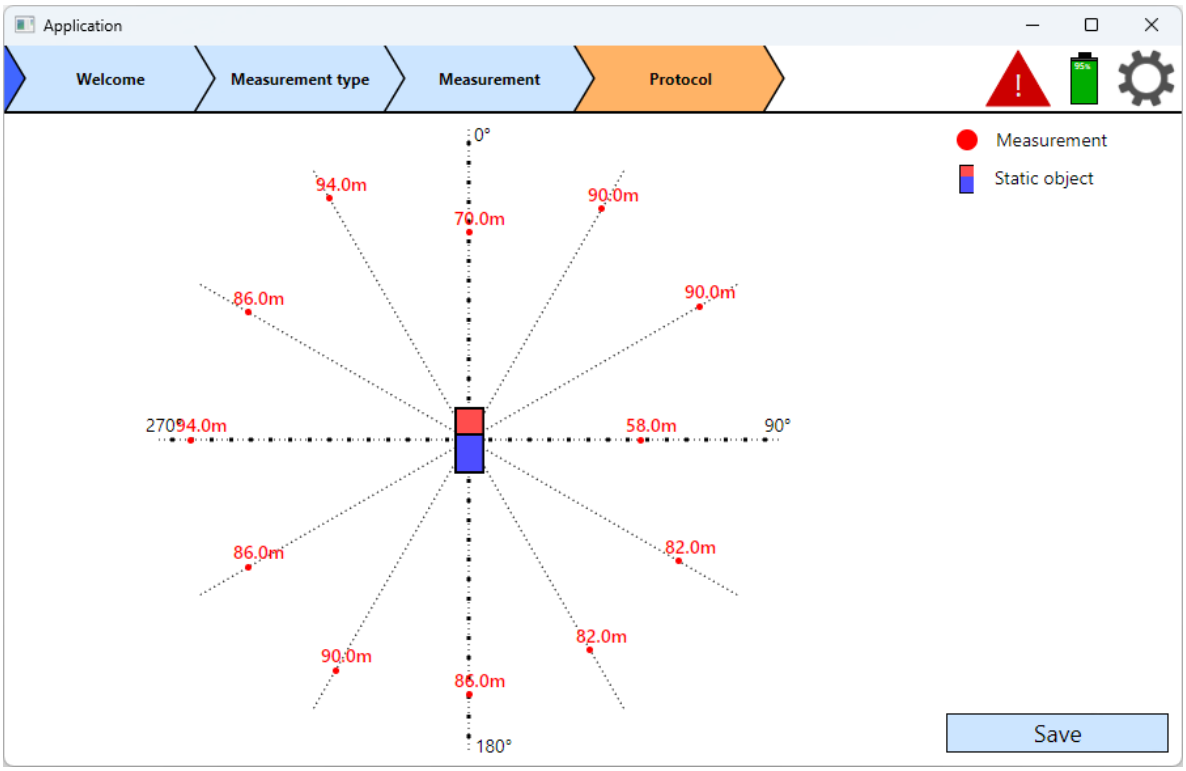


Figure 4.12: A screenshot of the *Main window* showing the *Protocol screen* specific to the *Key press* test scenario.

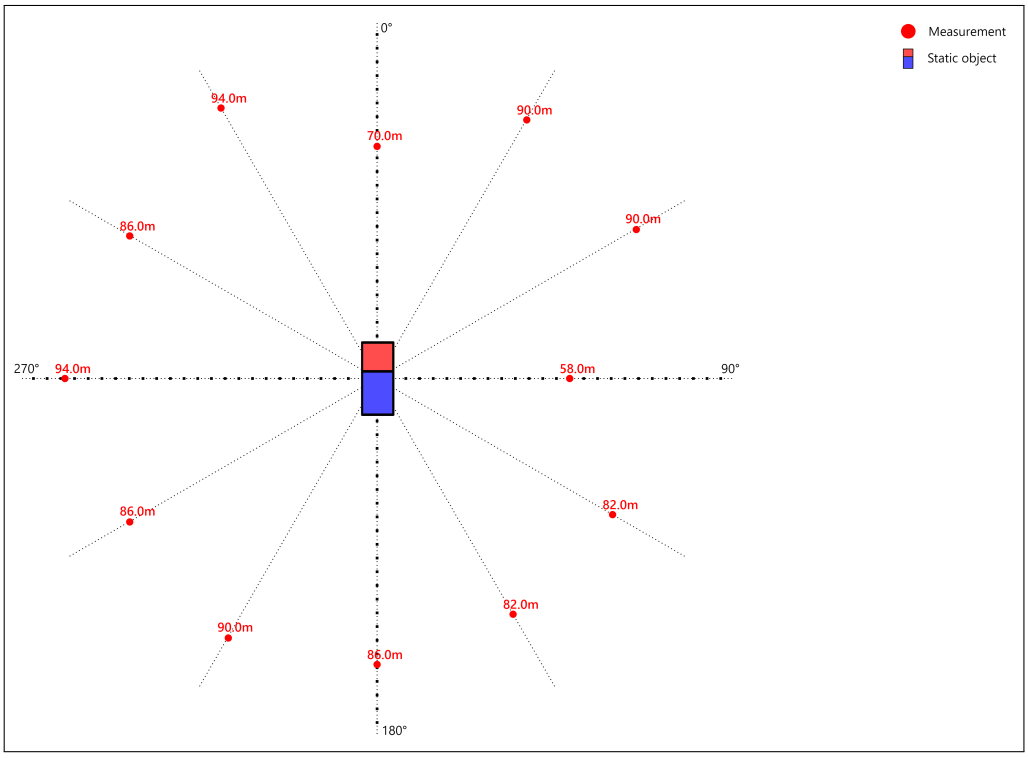


Figure 4.13: An image generated by the Operating software from the *Protocol screen* specific to the *Key press* scenario showing measurement results.

Settings window

The *Settings window* is opened by clicking the cogwheel icon in the *Header bar*. In addition to the location used to store the test results, the *Settings window* is used to set the folder where the test progress is saved when the Operator clicks the *Save progress* button on the *Measurement screen*. Further, the IP address and TCP ports used to connect the Operating software to the Robot are displayed in the *Settings window*; however, these values are not editable. A screenshot of the *Settings window* is available in Fig. 4.14.

4.1.3 Robot navigation

The Operating software navigates the Robot through the test by sending *waypoints* and commands that adjust the Robot's state, i.e., the height of the Remote key or whether a button is pressed.

We assume a waypoint consists of three components: X and Y coordinates and a heading defining the final rotation of the Robot. All these values are in the *Car coordinate system*. The Car coordinate system is a right-handed coordinate system that originates in the Car's centre and is measured in metres. The X coordinate increases when moving in front of the Car and decreases behind the Car. The Y coordinate increases to the Car's left and decreases to the right. The angle is in radians and is measured from the X-axis in a counterclockwise orientation.

The Robot informs the Operating software once a waypoint is reached. If the Robot should be stopped before reaching the sent waypoint, which may happen when the Operator

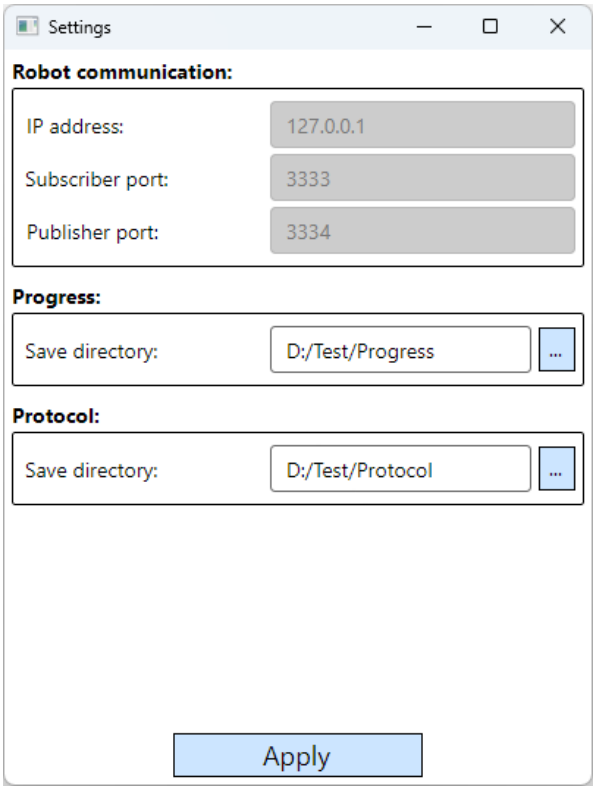


Figure 4.14: A screenshot of the *Settings window*.

presses the *Stop* button or when the Receiver receives the correct signal in the *Autolock* test scenario, a *Stop* command is sent to the Robot. As a result, the Robot stops following the waypoint and comes to a complete stop. Instead of sending a *Stop* command, a new waypoint may also be sent to the Robot to command the Robot to follow the new waypoint instead of the previous one, which is also used in the *Autolock* test scenario.

Pseudocodes showing waypoint generation for the *Key press* and *Autolock* test scenarios are available in Algorithm 1 and Algorithm 2, respectively. The algorithms shown have been simplified to highlight their primary flows.

4.1.4 Robot communication

As discussed, the Operating software communicates with the Robot using the ZeroMQ library's publisher-subscriber model, built on top of TCP, with messages serialised using Protocol Buffers (protobuf). The Operating software's outbound communication is detailed in Section 4.1.4.1, while the inbound communication is in Section 4.1.4.2. The protobuf serialisation format is further discussed in Section 4.1.4.3.

4.1.4.1 Outbound communication

The Operating software holds a reference to a *Command* object that contains information about the state to which the Robot should transition, e.g., a waypoint toward which the Robot should move, the height of the Remote key, or the speed at which the Robot should travel.

Algorithm 1 *Key press* scenario waypoint generation.

Require: $d_{min} \in \mathbb{R}_0^+$, $d_{max} \in [d_{min}, +\infty)$, $d_{str} \in [0, d_{max} - d_{min}]$, $d_{app} \in (0, +\infty)$, $\varphi_{off} \in (0, 2\pi]$

$\varphi \leftarrow 0$
while $|\varphi| < 2\pi$ **do**
  $v_{ax} \leftarrow (\cos\varphi, \sin\varphi)$ ▷ Axis vector of unit length
  $d \leftarrow d_{max}$
  SendWaypoint($v_{ax} \cdot d$, $\varphi + \pi$) ▷ Send waypoint (including heading) to Robot
  WaitUntilWaypointReached() ▷ Information received from Robot
  $d \leftarrow d - d_{str}$ ▷ Approach by the *Straightening distance*
  SendWaypoint($v_{ax} \cdot d$, $\varphi + \pi$)
  WaitUntilWaypointReached()
  while $d \geq d_{min}$ **do**
    SendPressButtonCommand()
    $r_{CAN} \leftarrow$ ReadStateFromCAN() ▷ Get the state from CAN bus.
    if r_{CAN} is OK **then**
      break ▷ On success, finish current axis
    end if
    $d \leftarrow d - d_{app}$
  end while
  $\varphi \leftarrow \varphi - \varphi_{off}$ ▷ Move clockwise to the next axis
end while

Algorithm 2 *Autolock* scenario waypoint generation.

Require: $d_{min} \in \mathbb{R}_0^+$, $d_{max} \in (d_{min}, +\infty)$, $d_{str} \in [0, d_{max} - d_{min})$, $d_{mes} \in (0, +\infty)$

while HasNextAxis() **do**
  $(v_{off}, v_{dir}) \leftarrow$ GetNextAxis() ▷ Get a point and direction defining the axis
  $\varphi \leftarrow atan2(v_{dir}[1], v_{dir}[0]) + \pi$ ▷ Calculate the required heading
  $d \leftarrow d_{max}$
  SendWaypoint($v_{off} + v_{dir} \cdot d$, φ) ▷ Go to the maximum distance
  WaitUntilWaypointReached()
  $d \leftarrow d_{max} - d_{str}$
  SendWaypoint($v_{off} + v_{dir} \cdot d$, φ) ▷ Approach by the *Straightening distance*
  WaitUntilWaypointReached()
  $d \leftarrow d_{min}$
  SendWaypoint($v_{off} + v_{dir} \cdot d$, φ)
  WaitUntilWaypointReachedOrUnlockedState() ▷ Measure the unlock area
  $d \leftarrow d_{max} - d_{str}$
  SendWaypoint($v_{off} + v_{dir} \cdot d$, φ)
  WaitUntilWaypointReachedOrLockedState() ▷ Measure the lock area
end while

```
1     private async Task Send(CancellationToken cancellationToken, Action<Command>? mutationFunc)
2     {
3         if (!IsConnected)
4         {
5             throw new InvalidOperationException($"{nameof(Connect)}() must be called before
                  sending messages.");
6         }
7
8         await Task.Run(() =>
9         {
10            byte[] message;
11
12            lock (_command)
13            {
14                if (mutationFunc != null)
15                {
16                    mutationFunc.Invoke(_command);
17                    _command.Id++;
18                }
19
20                message = _command.ToByteArray();
21            }
22
23            lock (_zmqPublisher)
24            {
25                _zmqPublisher.SendFrame(message);
26            }
27        }, cancellationToken).ConfigureAwait(false);
28    }
```

Listing 4.7: C# code responsible for sending the *Command* object from the Operating software to the Robot.

The *Command* object is periodically sent to the Robot, so if the TCP connection to the Robot is lost or the Robot reboots, the Operating software needs not to be informed, and the system continues to function as intended. The Operating software mutates the *Command* object to navigate the Robot through the test scenario. Whenever the *Command* object is mutated, a new unique identifier is assigned to the object, which the Robot then uses to determine if the command has been received before. After mutating the *Command*, it is sent to the Robot without waiting for the next sending period. A C# *lock statement* is used to ensure that the *Command* object can be safely mutated from multiple threads; therefore, only one thread can access the *Command* object at a time.

A code snippet showing how the *Command* object is sent from the Operating software is available in Listing 4.7. In the implementation shown in the snippet, the *Command* object is named *_command*. The *_zmqPublisher* object is an instance of the *PublisherSocket* class provided by the ZeroMQ messaging library. It is also protected by a *lock statement* to prevent conflicts when multiple messages are sent from the same TCP socket.

Each waypoint sent from the Operating software also contains a unique identifier, which is then sent back from the Robot to the Operating software to confirm that the waypoint has been reached.

When the Operating software requires the Robot to press a button on the Remote key, an additional unique identifier is assigned to the operation so the Robot does not press the

button multiple times. Once the Robot confirms, using the identifier, that the button has been pressed, the Operating software removes the button-press operation from the *Command* object.

4.1.4.2 Inbound communication

Information from the Robot to the Operating software is sent in a *Status* message. The *Status* message is sent periodically by the Robot and contains data about the Robot's state, in addition to the confirmation identifiers mentioned in Section 4.1.4.1. The data includes the Robot's position relative to the Car, the Robot's heading, height of the Remote key, buttons pressed on the Remote key, battery charge, measured dimensions of the Car, logging messages, and notifications that should be displayed to the Operator under the warning triangle in the *Header bar*, Fig. 4.2.

Whenever a *Status* message is received, an event is raised, and all subscribed components are notified. Thus, the Operating software can react accordingly by updating the UI or sending a new command to the Robot.

4.1.4.3 Serialisation format

The two communicating parties run in different environments and are written in different programming languages, with the Operating software in C# and the Robot's Dispatcher node in Python; therefore, the sent messages must be serialised into a format that both can understand. Common text serialisation formats include JavaScript Object Notation (JSON)[15] and XML, which also have the advantage of being easily readable by humans. However, the main disadvantage of text formats is their inefficiency since they require more memory and serialisation/deserialisation time than would be necessary for machine-to-machine communication.

Therefore, we decided to use a binary serialisation format called protobuf[16], which is developed by Google and is currently free and open source. Google also provides a code generator that accepts a schema defined using protobuf's description language and outputs code in several programming languages, including C#, Python, C++, Rust, Java, JavaScript, and many others. The generated code contains class definitions in the specific languages and handles the serialisation/deserialisation of instances of the generated class. A snippet showing a protobuf definition of a message that contains data about the Robot's state is available in Listing 4.8. For more information on protobuf and its performance, see [13] and [14].

4.1.5 Car communication

The Operating software retrieves the current state of the Car's locks via the Car's CAN bus. CAN is a standard communication bus used in the automotive industry that has a deterministic arbitration mechanism to resolve conflicts when multiple devices, called nodes, attempt to send data simultaneously. The mechanism utilises a special wiring that ensures that logical zero bits dominate logical one bits, meaning that when a logical zero and a logical one are sent simultaneously, only the logical zero is preserved on the bus and read by the receiving nodes. As each message begins with an identifier, the message with the smallest

[15] https://www.json.org/
[16] https://protobuf.dev/

```
message RobotState {
  optional float x = 1;
  optional float y = 2;
  optional float heading = 3;
  optional float speed = 4;

  optional float deviceHeight = 5;
  repeated uint32 pressedDeviceButtons = 6;

  optional int32 localizationAccuracy = 7;
  optional uint32 batteryLevel = 8;

  optional bool isMappingCar = 9;
}
```

Listing 4.8: A schema defining a protobuf message containing information about the Robot's state.

identifier (when interpreted as an integer) will be selected during the arbitration phase for transmission over the CAN bus. Note that the identifier is associated with the message, not with the node sending the message. Thus, a single node may be able to send multiple messages with different identifiers, where the identifiers define the message's priority on the entire bus.

To connect the Operating software to the Car's CAN bus, we decided to use an adapter developed by the company PEAK-System[17], which connects to the CAN bus on one end and offers a Universal Serial Bus (USB) port on the other end. PEAK-System also provides a .NET library[18] for working with the CAN adapter.

A code snippet showing how the messages are read and processed by the Operating software is available in Listing 4.9. The *_worker* invokes the registered method, *OnMessageAvailable*, whenever a message becomes available in the *_worker*'s queue. The read message is then passed to the *ParseMessage()* method, which is implemented in derived classes to make it possible to handle different test scenarios. The parsed message is then passed to methods subscribed to the *OnMessageReceived* event.

4.2 Robot

The Robot carries the Remote key and presses the buttons on the Remote key to invoke communication between the Remote key and the Receiver. A description of the Robot's hardware is provided in Section 4.2.1, followed by the Robot's software in Section 4.2.2.

4.2.1 Hardware

The Robot is a four-wheeled ground vehicle with a differential drive, based on the Jackal platform from Clearpath Robotics[19]. Differential drive implies that the Robot's wheels cannot be steered; however, the Robot has two motors, each connected to the wheels on one side of the Robot, which allows the wheels on the left side of the Robot to be driven independently

[17] https://www.peak-system.com/
[18] https://docs.peak-system.com/API/PCAN-Basic.Net/html/624ccd00-c8b0-4199-96df-e9413e83e6ad.htm
[19] https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/

```
1       private readonly object _messageLock = new();
2       protected readonly Peak.Can.Basic.Worker _worker;
3
4       public event Action<TMessage>? OnMessageReceived;
5
6       public PCanBasicWorker()
7       {
8           _worker = new(PcanChannel.Usb01, GetBitrate());
9           _worker.MessageAvailable += OnMessageAvailable;
10      }
11
12      protected abstract TMessage? ParseMessage(PcanMessage message);
13
14      private void OnMessageAvailable(object? sender, MessageAvailableEventArgs e)
15      {
16          PcanMessage? message;
17
18          lock (_messageLock)
19          {
20              if (!_worker.Dequeue(out message, out _))
21              {
22                  message = null;
23              }
24          }
25
26          if (message != null)
27          {
28              var parsedMessage = ParseMessage(message);
29
30              if (parsedMessage != null)
31              {
32                  OnMessageReceived?.Invoke(parsedMessage);
33              }
34          }
35      }
```

Listing 4.9: C# code snippet showing the retrieval of messages from the Car's CAN bus using the PCAN-Basic .NET library provided by PEAK-System.

of the wheels on the right side of the Robot, rotating the Robot. For more information on differential drive steering and kinematics, see [15] and [16].

The Robot is equipped with multiple sensors to enable autonomous operation. The Global Positioning System (GPS) module provides the Robot with global position coordinates, although its accuracy decreases dramatically in indoor environments, as discussed in [17] or [18]. The Robot is further equipped with an Inertial Measurement Unit (IMU) and collects odometry data from its wheels to improve positioning accuracy and allow indoor use. Both the IMU and the odometry data provide localisation relative to a previously known position, which suffers from poor performance as measurement error accumulates over long periods. In outdoor environments, data from the GPS can correct the accumulated error. For indoor scenarios, a Light Detection and Ranging (LiDAR) is mounted on the Robot.

A LiDAR works by shooting light rays and measuring the time it takes for the rays to return, using the constant speed of light to calculate the distance, see [19]. Thus, LiDAR can be used not only for localisation relative to surrounding objects but also to measure distances from surrounding objects and to facilitate object detection, e.g. [20], [21]. As it would be impractical for the Operator to manually provide the Robot with information about the Car's position, the LiDAR is further utilised to detect the Car in the Robot's vicinity.

The Remote key is placed on top of a linear actuator, which is positioned vertically and allows the Robot to move the Remote key up and down as per the required height of the Remote key received from the Operating software. Alongside the Remote key are two servomotors used to press the lock and unlock buttons on the Remote key. Several LEDs indicate the Robot's state to the Operator and other personnel.

The peripherals, i.e., the linear actuator, the servomotors, and the LEDs, are not connected directly to the Robot's main computer; instead, they are connected to a dedicated microcontroller, which communicates with the main computer over a serial port. Communication with the Operating software is facilitated by a Wi-Fi module.

4.2.2 Software

The software running on the Robot can be decomposed into several parts. In Section 4.2.2.1, we look at the Robot Operating System (ROS), a framework on which our software is built. Section 4.2.2.2 follows with a description of the Robot's state machine, which determines the Robot's high-level behaviour. Then, communication with the Operating software is described in Section 4.2.2.3. Lastly, interaction with the Robot's peripherals is detailed in Section 4.2.2.4.

4.2.2.1 Robot Operating System

The Robot's software runs on ROS. Unlike what the name implies, ROS is rather a framework than an operating system. ROS provides tools and services for communication between individual application components, called nodes, and provides an infrastructure to integrate third-party nodes.

The inter-node communication in ROS is based on the publisher-subscriber model. Each node is allowed to publish messages on any number of topics, with each topic identified by a unique name. Other nodes then subscribe to the published topics using the topic's name.

Each topic has an associated message type, so only messages of the specified type can be sent by the publishers and thus read by the subscribers.

In addition to the many-to-many publisher-subscriber communication model via ROS topics, ROS also provides ROS services, which allow one-to-one communication in which one node sends a request message, and the other node responds with a reply message. The services are also identified by unique names.

Although communication logically works via topics and services, physically, sending data from one node to another is necessary. Therefore, a node must be able to identify the other communicating node by the name of the topic or the service. ROS solves this by designating one node as the ROS Master. All topics and services are registered with the Master. When a node needs to connect to another node for communication, it first queries the Master, which locates the corresponding node by the name of the topic or the service. The communication is then carried out directly between the two nodes, meaning the messages are not sent through the Master. For more information regarding ROS, see [22] and [23].

4.2.2.2 State machine

Before the Robot can start the test, it needs to localise itself, map its surroundings, and identify the Car, as it is the centre of the coordinate system used for sending waypoints by the Operating software. Therefore, the Operator is required to position the Robot at a predetermined location relative to the Car. This positioning is expected to be inaccurate; however, it should be sufficient to help the Robot identify the Car. The Car locator node then corrects the position of the Car. If required, the Car locator node may also create a few artificial waypoints around the Car, which the Robot follows to gather more information about the Car's dimensions. The Robot then uses a Simultaneous Localization And Mapping (SLAM) algorithm to create a map of its surrounding area and to localise itself within this map.

The Robot's state machine determines the high-level behaviour of the Robot. A state in the state machine determines the Robot's response to commands received from the Operating software.

Firstly, the Robot is in the *Starting* state and waits for all ROS nodes to initialise. Once the initialisation is complete, the state changes to *Ready to map car*, where the robot waits for a command from the Operating software. As the mapping of the Car is an internal process of the Robot, the Operating software does not have any specific commands to influence it. However, the Operating software is aware of the Robot mapping the Car, as the Robot sends this information to the Operating software.

Secondly, when a waypoint is received from the Operating software, the Robot starts mapping the Car. The Operator can stop the Robot's movement using the *Stop button*, which sends a stop command to the Robot. When the Operator resumes the test, the waypoint is re-sent to the Robot, and the Car mapping continues.

Lastly, once the Robot has successfully mapped the Car, it transitions to the *Following waypoint* state, wherein the Robot moves toward waypoints received from the Operating software. Similarly to when the Car was being mapped, a stop command makes the Robot stop, whereas a new waypoint resumes the Robot's movement. A visualisation of the Robot's state machine is in Fig. 4.15.

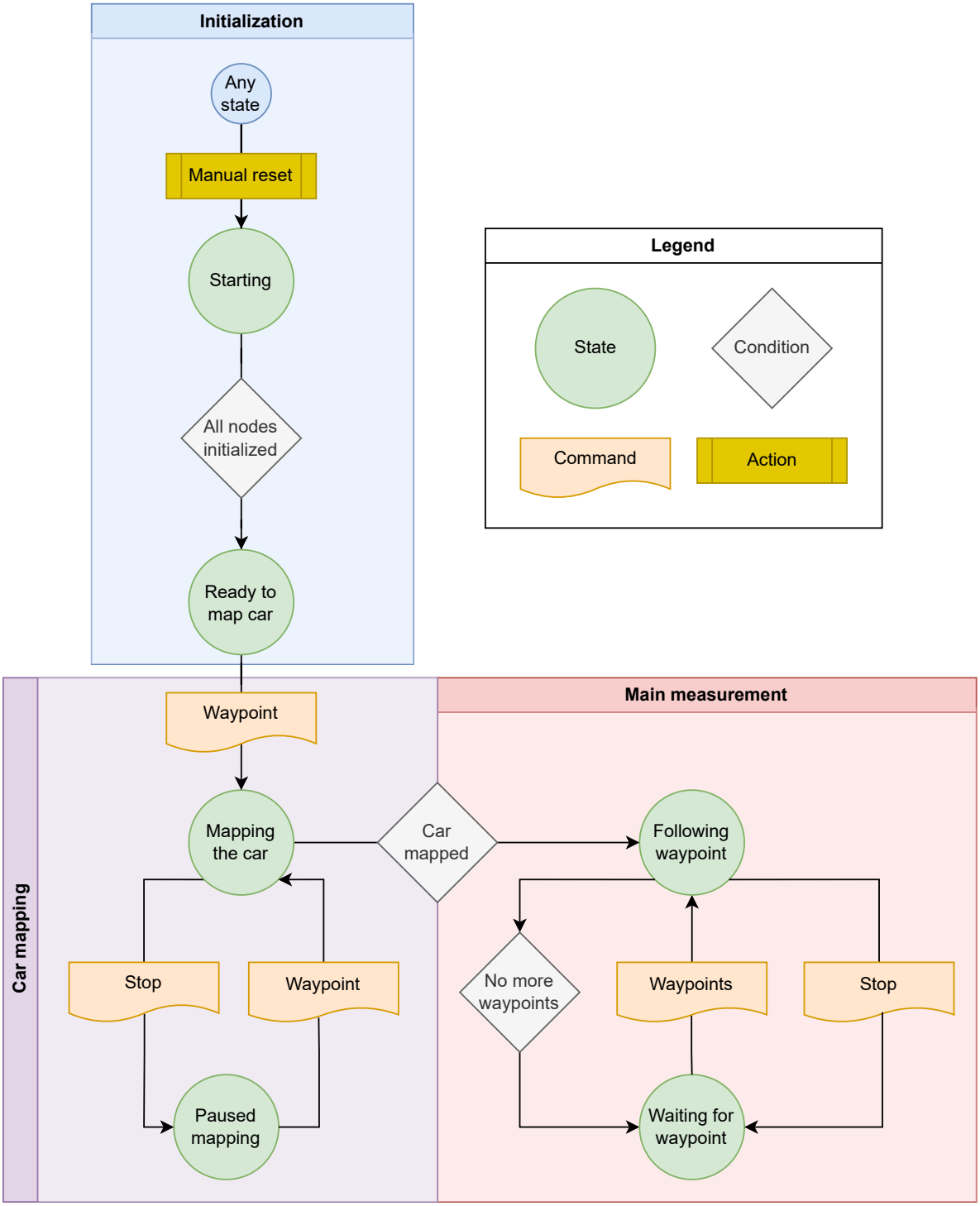


Figure 4.15: A depiction of the Robot's state machine.

4.2.2.3 Operating software communication

As described in Section 4.1.4, the Robot communicates with the Operating software through the ZeroMQ messaging library, which provides an implementation of the publisher-subscriber communication model over TCP. The messages sent through ZeroMQ are serialised using protobuf.

The Dispatcher node, written in Python[20], handles communication with the Operating software. Python is a popular dynamically typed programming language with automatic garbage collection often used in robotics and machine learning development due to its vast standard library and ready-to-use software components developed by members of its community. Applications written in Python are not compiled into machine instructions but executed by a Python interpreter.

Whenever a command is received from the Operating software, the Dispatcher node translates the command into actions to be performed by other ROS nodes on the Robot, e.g., delegates commands to the motion planner to move toward a received waypoint or instructs the lifting mechanism to set the height of the Remote key. Additionally, the Dispatcher node collects information, such as the position of the Robot or the state of the Robot's battery, from other nodes on the Robot and sends the information to the Operating software, which can use it in the test flow or to inform the Operator about the Robot's state.

4.2.2.4 Peripherals microcontroller

As mentioned in Section 4.2.1, the peripherals connected to the Robot are controlled by a dedicated microcontroller instead of the Robot's main computer. The microcontroller is connected to the Robot's main computer using a serial port. To integrate the microcontroller into the ROS infrastructure, we use rosserial[21].

Rosserial comprises two parts, one on the microcontroller and the other on the connected computer. These two parts communicate with each other over the serial port. The part on the computer is a ROS node, which is subscribed to all topics required by the microcontroller. Whenever a message on a subscribed topic is received, it is serialised and sent to the microcontroller. Conversely, when the microcontroller needs to publish a message, it sends the message through the serial port to the ROS node, which then publishes the message on a ROS topic. For more information on using ROS with embedded devices, see [24].

[20] https://www.python.org/
[21] https://wiki.ros.org/rosserial

Chapter 5

Testing and evaluation

Throughout software development, it is essential to prepare, execute and evaluate tests that verify whether the developed system meets the specified design goals and behaves as expected. Acceptance testing is often used to check whether a newly added feature works as intended and, thus, can be accepted. In many situations, acceptance testing can be performed manually, as it is limited in scale to only the new feature and, if successful, needs only to be performed once per feature.

However, in software development, a new feature, a bug fix, or a change in the environment (e.g., an operating system update) may break a seemingly unrelated part of the application. Catching such errors is called regression testing, and performing it manually would require a lot of human resources, as the entire application would need to be tested after every change in the codebase or the environment. Therefore, tests are automated by software developers. Although the process of making an automatic test usually takes longer than executing the test manually, once developed, the test can be run repeatedly without human interaction.

Tests can also be distinguished by their scope. Unit tests verify the correctness of a single unit, e.g., a function or a class. They usually perform only simple operations and are quick to execute; therefore, many unit tests can be executed in a limited time, making them suitable for running frequently.

When more units are put together, and their interaction and cooperation are verified, it is called an integration test. Integration tests often take longer to execute because they work with a larger part of the system. Note that the units part of an integration test should also be tested independently using unit tests.

Finally, system tests evaluate the correctness of the entire system. A system test is meant to be as close to the actual usage of the system as possible. Such tests are necessary before deployment to end users; however, they might take a long time to run or can be costly to execute, so they are often used sparingly.

Several testing frameworks have been developed to facilitate the development of automated tests. A programming language is often associated with a few popular unit testing frameworks, although more may be available. In C#, MSTest, NUnit, and xUnit are the most common; in Python, unittest is popular, while in Java, JUnit, TestNG, Cucumber, and many others are used. Some testing frameworks support multiple languages, such as Selenium[1], which specialises in testing web applications. For more information on software testing, see [25].

[1] https://www.selenium.dev/

```
1  [Fact(Timeout = 10_000)]
2  public void Connect_Success()
3  {
4      _robotCommunication.Connect("127.0.0.1", ZmqServerFixture.PublisherPort,
           ZmqServerFixture.SubscriberPort);
5
6      Assert.True(_robotCommunication.IsConnected);
7  }
```

Listing 5.1: C# code used to test a successful connection attempt using the *Connect* method.

5.1 Unit testing

For testing the Operating software, which is written in C#, we have selected the xUnit[2] testing framework. We demonstrate the usage of xUnit in a few unit tests that verify the functioning of the *Connect* method, shown in Listing 4.2, used to connect to the Robot. The *Connect* method establishes a connection using the ZeroMQ library, which does not let us know if the connection has truly been established. Therefore, the *Connect* method first checks if a TCP connection can be established using the specified ports. To verify the behaviour of the *Connect* method, we need to check that the method fails when invalid parameters are supplied, i.e., an IP address and TCP ports that do not have a counterparty to connect to or that are not in the correct format, and that the method does not fail with valid parameters.

The second case is simple, as it requires us to call the *Connect* method using valid parameters, of which there is only one possible combination. The *IsConnected* property then lets us know if the method has been successful. The unit test implementation is in Listing 5.1. The first line, containing the *Fact* attribute, informs the xUnit framework that the attached method is a test. Furthermore, the attribute's *Timeout* parameter specifies the time after which the test should automatically be terminated by xUnit and considered as failed, which is used to catch cases in which the *Connect* method's timeout fails. Line 4 contains the call to the *Connect* method with valid parameters. Finally, line 6 uses the xUnit's *Assert* class to check whether the provided variable, in our case *IsConnected*, resolves to true.

Verifying the first case, with invalid parameters, is more involved, as we need to check that it fails when any of the provided parameters are incorrect. The *Connect* method has three parameters, each of them can be valid, invalid, or have an incorrect format, resulting in $3^3 = 27$ possible combinations, of which one is valid and the rest is invalid. In addition to the *Fact* attribute, xUnit also offers the *Theory* attribute, which indicates a test method with parameters. Similarly to the previous situation, a *Timeout* parameter is present on the attribute to catch cases where the *Connect* method's timeout fails. We could then use twenty-six *InlineData* attributes, each specifying a different combination of the parameters. A different approach is to use xUnit's *MemberData* attribute instead of the *InlineData* attribute, which allows us to specify a field, property, or method that provides the parameters. The test implementation is shown in Listing 5.2. Note line 32 that excludes the valid combination of the parameters. Lines 9 through 15 catch the expected possible exceptions that the *Connect* method might throw. The *AggregateException*s can be thrown as a result of an exception raised when executing a function passed to the *Task.Run* method in the *Connect* method's body. Line 17 asserts that the connection has not been established.

[2] https://xunit.net/

```
1  [Theory(Timeout = 10_000)]
2  [MemberData(nameof(Get_Connect_Fail_Combinations))]
3  public void Connect_Fail(string ipAddress, int subscriberPort, int publisherPort)
4  {
5      try
6      {
7          _robotCommunication.Connect(ipAddress, subscriberPort, publisherPort);
8      }
9      catch (TimeoutException) { }
10     catch (SocketException) { }
11     catch (AggregateException ex) when (ex.InnerExceptions.Any(x => x.GetType() ==
           typeof(SocketException))) { }
12     catch (ArgumentOutOfRangeException) { }
13     catch (AggregateException ex) when (ex.InnerExceptions.Any(x => x.GetType() ==
           typeof(ArgumentOutOfRangeException))) { }
14     catch (FormatException) { }
15     catch (AggregateException ex) when (ex.InnerExceptions.Any(x => x.GetType() ==
           typeof(FormatException))) { }
16
17     Assert.False(_robotCommunication.IsConnected);
18 }
19
20 public static IEnumerable<object[]> Get_Connect_Fail_Combinations()
21 {
22     string[] ipAddresses = new[] { "127.0.0.1", "127.0.0.2", "256.0.0.1" };
23     int[] subscriberPorts = new[] { ZmqServerFixture.SubscriberPort,
           ZmqServerFixture.SubscriberPort - 1, -1 };
24     int[] publisherPorts = new[] { ZmqServerFixture.PublisherPort,
           ZmqServerFixture.PublisherPort + 1, -1 };
25
26     foreach (var ipAddress in ipAddresses)
27     {
28         foreach (var subscriberPort in subscriberPorts)
29         {
30             foreach (var publisherPort in publisherPorts)
31             {
32                 if (ipAddress == ipAddresses[0] && subscriberPort == subscriberPorts[0] &&
                       publisherPort == publisherPorts[0])
33                     continue;
34
35                 yield return new object[] { ipAddress, subscriberPort, publisherPort };
36             }
37         }
38     }
39 }
```

Listing 5.2: C# code used to test unsuccessful connection attempts using the *Connect* method.

Both presented unit tests work with a variable named *_robotCommunication*, which contains a reference to an instance of the *RobotCommunication* class responsible for handling communication with the Robot. The instance is constructed in the constructor of the class that contains the unit tests. The xUnit framework invokes the constructor before each test run; thus, each unit test has its own instance of the *RobotCommunication* class. The constructor of the *RobotCommunication* class requires three parameters - a logger, a configuration provider, and a mapper which maps protobuf messages to internally used models. Creating an instance of the logger and the configuration provider would require additional dependencies to be attached to the tests. Therefore, we decided to mock them instead.

5.2 Mocking

Mocking is a programming technique used to isolate tests by replacing their dependencies with fake objects, called mock objects, which simulate the behaviour of the actual dependencies. In unit testing, mocking is often used when a unit relies on external services not directly under the developer's control, making it difficult to predict their state during test runs or when the dependencies are problematic to set up. Additionally, mocking decouples dependent units so that they can be tested in isolation, which is one of the goals of unit testing. Mocking frameworks are often available for popular programming languages, such as the Mockito framework for Java, unittest.mock for Python, or Moq for C#. More information on mocking can be found in [26].

For our tests, we used the Moq[3] mocking framework. In Listing 5.3, we create a mock object for the logger and the configuration provider. Furthermore, the configuration provider is configured to return the value 1000 when its *CommandRepeatInterval* property is accessed and the value 2000 when *ConnectionAttemptTimeout* is accessed. These mock objects, along with a *RobotCommunicationMapper*, are then passed to the *RobotCommunication*'s constructor to create the instance of the *RobotCommunication* class that is used in the unit tests.

5.3 Connection server

So far, we have created an instance of the *RobotCommunication* class that we want to test and have prepared unit tests to verify the functionality of the *Connect* method. One last problem remains to be solved: we need a Robot to connect to. Or, rather, some other server that can be launched alongside the unit tests and would act as the Robot, so we would not need to start the Robot every time we needed to run tests on the *Connect* method. This server could be launched just once at the start of the tests and then terminated once all the tests have finished. For such purposes, xUnit allows developers to group tests into collections with common fixture data that are initialised before the first test from the collection is run and disposed of after the last test is executed.

The fixture used to simulate the Robot's server is shown in Listing 5.4. The collection must then be defined, as shown in Listing 5.5. Finally, a class containing unit tests that belong to the defined collection must be marked using the *Collection* attribute, as in Listing 5.3.

[3] https://github.com/devlooped/moq

```
1  [Collection(ZmqServerCollection.Name)]
2  public class Connect : IDisposable
3  {
4      private readonly IRobotCommunication _robotCommunication;
5
6      public Connect()
7      {
8          var loggerMock = Moq.Mock.Of<ILogger>();
9
10         var configurationMock = Moq.Mock.Of<IRobotCommunicationConfiguration>(conf =>
11             conf.CommandRepeatInterval == 1000 &&
12             conf.ConnectionAttemptTimeout == 2000);
13
14         _robotCommunication = new Communication.RobotCommunication(
15             loggerMock,
16             configurationMock,
17             new RobotCommunicationMapper()
18         );
19     }
20
21     // ---- unit tests below ----
22 }
```

Listing 5.3: C# code used prepare an instance of the *RobotCommunication* class for unit testing.

```
1  public class ZmqServerFixture : IDisposable
2  {
3      public const int SubscriberPort = 3333;
4
5      public const int PublisherPort = 3334;
6
7      private readonly SubscriberSocket _subscriberSocket;
8
9      private readonly PublisherSocket _publisherSocket;
10
11     public ZmqServerFixture()
12     {
13         _subscriberSocket = new SubscriberSocket($"@tcp://127.0.0.1:{SubscriberPort}");
14         _publisherSocket = new PublisherSocket($"@tcp://127.0.0.1:{PublisherPort}");
15     }
16
17     public void Dispose()
18     {
19         _subscriberSocket.Dispose();
20         _publisherSocket.Dispose();
21     }
22 }
```

Listing 5.4: C# code showing a data fixture used to simulate the Robot's server for unit testing.

```
1 [CollectionDefinition(Name)]
2 public class ZmqServerCollection : ICollectionFixture<ZmqServerFixture>
3 {
4     public const string Name = "ZMQ server collection";
5 }
```

Listing 5.5: C# code showing the definition of a collection that uses *ZmqServerFixture* as its data fixture.

5.4 Integration testing

In addition to unit tests, we used xUnit to implement integration tests verifying that the Robot reacts appropriately to received commands. In Listing 5.6, we present the code used to verify that the Robot can change the Remote key's height and inform the Operating software about such change. The *AutoResetEvent*s, declared on lines 19 and 20, are used to put the running thread to sleep on lines 33 and 44, respectively. When the *AutoResetEvent*'s *WaitOne* method returns true, it means that another thread has called the *AutoResetEvent*'s *Set* method. If the waiting thread is woken up otherwise, the *WaitOne* method returns false. Before the thread is put to sleep, an anonymous function is subscribed to the *RobotCommunication*'s *StateChanged* event, which is invoked each time the Operating software receives a status message from the Robot. The first function, defined on lines 22 to 29, wakes the thread waiting on line 33 once the height received from the Robot is within a 10% tolerance of the required height. The required height is sent to the Robot on line 31. The second anonymous function, defined on lines 35 to 42, is used to wake the thread waiting on line 44, which happens when the height received from the Robot exceeds the defined 10% tolerance threshold, leading to a test failure. If the thread waiting on line 44 is not woken up before a $5000ms$ timeout expires, the *AutoResetEvent*'s *WaitOne* method returns false, and the test succeeds.

Note the fixture passed as a constructor argument on line 8 in the Listing 5.6. As discussed in Section 4.1.4, a unique identifier is sent with each message. If the test method used a new *RobotCommunication* instance each time it was invoked, as was the case in the unit tests presented previously, the new instance would always use the initial identifier when sending the command on line 31. Unless the Robot was to be restarted before each test run, the *RobotCommunication* instance used must be the same for all test runs. To achieve this, we moved the *RobotCommunication* instance to the test data fixture, which xUnit provides us in the class constructor.

5.5 Tools for manual testing and development

Many parts of the Operating software require communication with the Robot. Starting the Robot each time a developer needed to test a change in the GUI or other parts of the application would be impractical. Moreover, if the Robot were to be used heavily throughout the development cycle of the Operating software, it could slow down the development of the Robot itself, as it would prevent the work on the Operating software and the Robot from being performed in parallel. Therefore, we developed a simple simulator that can be used in place of the Robot when developing the Operating software, further discussed in Section 5.5.1.

Additionally, both test scenarios presented in this thesis depend on communication between the Operating software and the Car. Section 5.5.2 discusses a basic substitute for the

```
1  [Collection(RobotCommunicationCollection.Name)]
2  public class SetHeight
3  {
4      private readonly IRobotCommunication _robotCommunication;
5
6      private const float AllowedDeviation = 0.1f;
7
8      public SetHeight(RobotCommunicationFixture fixture)
9      {
10         _robotCommunication = fixture.RobotCommunication;
11     }
12
13     [Theory(Timeout = 15_000)]
14     [InlineData(70)]
15     [InlineData(102.4)]
16     [InlineData(150)]
17     public async Task SetHeight_Success(float height)
18     {
19         AutoResetEvent heightSetEvent = new(false);
20         AutoResetEvent heightBrokenEvent = new(false);
21
22         _robotCommunication.StateChanged += state =>
23         {
24             if (state.DeviceHeight.HasValue
25                 && Math.Abs(state.DeviceHeight.Value - height) < AllowedDeviation * height)
26             {
27                 heightSetEvent.Set();
28             }
29         };
30
31         await _robotCommunication.SetDeviceHeight(height);
32
33         Assert.True(heightSetEvent.WaitOne());
34
35         _robotCommunication.StateChanged += state =>
36         {
37             if (state.DeviceHeight.HasValue
38                 && Math.Abs(state.DeviceHeight.Value - height) > AllowedDeviation * height)
39             {
40                 heightBrokenEvent.Set();
41             }
42         };
43
44         Assert.False(heightBrokenEvent.WaitOne(5_000));
45     }
46 }
```

Listing 5.6: C# code testing the Robot's ability to change the height of the Remote key.

Car, removing the need for an actual car during development.

Further, developing the Robot's software, such as the Dispatcher node and others, can also be performed without requiring the physical robot. To simulate the physics of the Robot, we use the Gazebo simulator[4], which is more detailed in Section 5.5.3.

Finally, a visualisation tool used in the ROS framework, called RViz, is discussed in Section 5.5.4.

5.5.1 Robot substitution

To substitute the Robot when developing the Operating software, an extension of the ZeroMQ server, presented in Listing 5.4, that serves as a simulator of the Robot's behaviour has been developed.

The simulator opens communication ports for the Operating software to connect to, much as the Robot would. Once connected, the simulator waits for messages from the Operating software. The messages are then deserialised from the protobuf format, and their execution is simulated. The simulator keeps track of the virtual state of the Robot and, when a waypoint is received, updates the Robot's virtual position based on the set speed to imitate its movement. Periodically, the simulator sends status messages to the Operating software to inform it of the Robot's position and other properties, such as the Robot's heading or the height of the Remote key.

Execution of other commands, such as the command to stop the Robot or to set the height of the Remote key, are simulated similarly by changing the state of the virtual Robot and sending periodic updates to the Operating software.

5.5.2 Car substitution

The Operating software requires a connection to the Car's CAN bus, making development more complicated and time-consuming. Therefore, we created a simple component that mimics the communication with the Car. The component implements the same interface as the actual CAN bus connection implementation, so the rest of the Operating software does not need to be altered to use it. The interface consists of methods that allow connecting/disconnecting to/from the CAN bus and an event that is raised whenever a relevant message is read on the bus.

Implemented as a timer-based mechanism, the substitute component initiates or suspends periodic callbacks upon invocation of the connection or disconnection methods. During active intervals, a callback function generates a random number, which is subsequently compared against a predefined threshold. Based on the comparison, the event on the CAN interface is either raised or suppressed. This design gives developers flexibility in adjusting the likelihood of the event being raised by modifying the threshold parameter.

5.5.3 Gazebo

In robotics development, it is essential to thoroughly test and validate the behaviour of a robot and its interaction with the surrounding environment before deploying it in real-world

[4] https://gazebosim.org/

scenarios. Such pre-deployment testing is crucial in the iterative development of the robot, as it facilitates the identification of potential flaws and ensures the safety and reliability of the robot's operation.

Real-world experiments that involve previously untested hardware or software components carry inherent risks. The complexity of robotic systems, coupled with the unpredictability of real-world environments, amplifies the likelihood of errors or malfunctions. These errors pose risks not only to the robot itself but also to the surrounding environment and individuals nearby. Using realistic simulation environments, such as Gazebo, developers can proactively address these risks, minimising the possibility of a severe error in real-world deployments.

The Gazebo simulator is a platform for creating realistic simulations of robots and their surrounding environments. It can be seamlessly integrated into ROS-based projects using dedicated ROS packages, which simplify the setup and enable data exchange between Gazebo simulations and ROS nodes, further improving and streamlining its capabilities in robotics development. For more information on Gazebo and its usage, see [27], [28], and [29].

5.5.4 RViz

RViz is a 3D visualisation tool designed for the ROS ecosystem. It allows users to visualise various information in a three-dimensional space. Since RViz was designed specifically for ROS, it integrates simply by subscribing to published ROS topics.

Therefore, apart from visualising the positions of objects and robots, one can also visualise virtual objects, such as waypoints or collision boundaries. Moreover, RViz can also publish to ROS topics and interact with ROS nodes through ROS services, allowing users to, for example, operate the robot from its visual UI. Additional information on RViz can be found in [30].

5.6 Real-world experiments

Despite the advancements made in simulation technologies such as Gazebo and the convenience of virtual environments for testing, the transition to real-world scenarios remains vital for validating the developed systems. While simulations offer a controlled and repeatable environment, the complexity and unpredictability of the real world often introduce unforeseen challenges and edge cases that may not have been accounted for during testing in virtual environments.

In our real-world experiments, particular emphasis was put on assessing the system's long-term stability to ensure that it maintains performance without accumulating excessive error over extended durations. Moreover, these experiments tested the performance and resilience of the proposed system in environments with imperfect communication channels, introducing delays and message failures between the Operating software and the Robot.

Additionally, the real-world experiments allowed us to validate the system's functionality when connected to a real car's CAN bus. This integration enabled us to verify the operation of the Operating software's CAN bus interface.

5.7 Evaluation

The implementation of automated regression testing methods helped us focus on new features and on iteratively improving the proposed system by providing safeguards that prevented us from accidentally breaking existing components. The writing of unit tests and integration tests proved crucial in the development of the system. Throughout development, the automated tests' effectiveness increased as the whole system's complexity increased.

The Robot simulator, discussed in Section 5.5.1, played a vital role in developing the Operating software. Especially in the early development phase, the Operating software expanded quicker than the Robot, making relying on their mutual interaction impossible. Using the simulator, we could decouple the development of the Operating software from the development of the Robot so that each could move forward at its own pace. However, once it became possible to connect the two parts, some aspects of the communication and interaction needed to be revised as a result of the independent development. Nevertheless, the required adjustments were of only minor complexity, and the overall contribution of the developed simulator greatly outweighed these disadvantages.

When developing the Robot, the Gazebo simulator proved effective in providing a realistic environment for testing various aspects of the system. New features could be quickly tested without deploying the Robot in real-world scenarios, which might take a long time to set up and risk damage to the physical hardware. Furthermore, using the RViz visualisation tool helped us debug errors in the Robot's software; for example, by visualising the currently followed waypoint, we managed to identify a problem where the Robot failed to switch to a newer waypoint if received before the first waypoint was reached.

On the other hand, while simulation environments offer fast and convenient testing conditions, they cannot accurately replicate certain real-world phenomena, such as sensor noise and environmental variability. Recognising these limitations, we conducted a series of real-world experiments focused on improving the system's robustness and resilience in actual operating conditions. Throughout these experiments, we encountered unforeseen challenges that would have remained undetected in simulation environments alone.

One notable issue that came to light during the deployment of the Robot was the failure of its Wi-Fi hotspot to initialise. This unexpected setback signified the importance of real-world testing, even as simulators become more advanced and realistic. By solving these new challenges, we iteratively refined the system to improve its reliability in real-world scenarios.

Regarding the system's performance, we deemed the design to be able to operate without substantial issues. Even at more considerable distances, when the connection between the Robot and the Operating software became error-prone, unstable, or broken, it was quickly re-established, and the system could seamlessly resume its operation. However, due to the significant weight and height of the linear actuator attached to the Robot, the Robot's centre of gravity shifted upward. To prevent the Robot from toppling when making rapid movements, we needed to enforce limits on the Robot's linear and angular accelerations.

All in all, the logistics and costs associated with real-world experiments posed challenges in terms of scalability and repeatability. Nonetheless, the experiments allowed us to observe the complete system in its final environment, successfully validating its possible use in the scenarios discussed in this thesis.

Chapter 6

Conclusion

In conclusion, this thesis presented the design, implementation, and possible applications of a mobile testing robot. A detailed explanation of the tools and technologies used in the development has been provided, along with sample code snippets showing their utilisation.

Several different systems, platforms, and frameworks have been explored to create the solution. The development of the Robot presented challenges in terms of its hardware and software, as we aimed for a general design that allowed for different possible scenarios. For implementing the Robot's software, we chose the ROS framework, which is widely used in robotics development. Thus, we could rely on ROS's extensive documentation and third-party packages to help us make the Robot. Additionally, thanks to the infrastructure provided by ROS, we could easily separate the Robot's logical components into individual nodes and use ROS's communication techniques to exchange data among them, allowing for a clean and maintainable codebase. The created Dispatcher node served as a gateway between the ROS ecosystem running on the Robot and the Operating software running on a separate personal computer.

We designed a few basic actions that the Robot needed to execute. The actions were then combined in the Operating software to create the flow of a test scenario and sent to the Robot as commands over a wireless network. Using this approach, we could limit the scope of the Robot's need to be aware of the specific test scenario.

To make the system accessible and easily adoptable by the Operator, we created a simplistic GUI that allows the Operator to interact with the Operating software, set its parameters, monitor the progress of the test, and gather the results. The GUI was built using Avalonia UI, which is a modern UI framework that enables the development of cross-platform UI applications using .NET. Although we sometimes found Avalonia UI's documentation lacking, its similarity to the well-established WPF framework allowed us to overcome any obstacles we faced. By continuously consulting the UI with its potential users, we avoided unnecessary design changes in later development phases.

Finally, to improve the quality and reliability of the developed system, we utilised various tools and techniques, including automated unit and integration testing, that helped us ensure the correctness of the system throughout development. Using the Gazebo simulator, we could realistically verify both the Robot and the Operating software without having to deploy the Robot in real-world situations, which would be both costly and time-consuming.

However, no simulation is perfect, so it is always necessary to perform real-world experiments. Further, as we performed the experiments relatively late in the development process, we discovered that some real-world tests should be performed as early as possible, without compromising safety, to allow for a better understanding of the difference between simulated and real-world behaviour of the system, which can prevent certain misconceptions about the qualities of the system that might seem promising in simulations but fail in the real world. For

example, the node responsible for motion planning had to be rewritten several times as we discovered that the Robot had problems performing some motions on certain surfaces, such as asphalt concrete, which could have been prevented had we run some experiments sooner.

The journey from conceptualisation to realisation of the mobile testing robot has been both challenging and rewarding, offering numerous learning opportunities and insights into the practical aspects of robotics development. As technology continues to advance, the potential applications for such robots are vast and varied, ranging across many present and future industries.

Chapter 7

References

[1] Z. Hanzálek, J. Záhora, and M. Sojka, "Cone slalom with automated sports car–trajectory planning algorithm," *IEEE Transactions on Vehicular Technology*, 2023.

[2] M. Mohanan and A. Salgoankar, "A survey of robotic motion planning in dynamic environments," *Robotics and Autonomous Systems*, vol. 100, pp. 171–185, 2018.

[3] J. Klapálek, M. Sojka, and Z. Hanzálek, "Comparison of control approaches for autonomous race car model," in *Proceedings of the FISITA 2021 World Congress*, FISITA-International Federation of Automotive Engineering Societies, 2021.

[4] M. Elbanhawi and M. Simic, "Sampling-based robot motion planning: A review," *Ieee access*, vol. 2, pp. 56–77, 2014.

[5] J. Klapálek, A. Novák, M. Sojka, and Z. Hanzálek, "Car racing line optimization with genetic algorithm using approximate homeomorphism," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2021, pp. 601–607.

[6] A. Kalyanaraman, Y. Zeng, S. Rakshit, and V. Jain, "Caraokey : Car states sensing via the ultra-wideband keyless infrastructure," in *2020 17th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, 2020, pp. 1–9. DOI: 10.1109/SECON48991.2020.9158440.

[7] K. Pahlavan and P. Krishnamurthy, "Evolution and impact of wi-fi technology and applications: A historical perspective," *International Journal of Wireless Information Networks*, vol. 28, pp. 3–19, 2021.

[8] E. Khorov, I. Levitsky, and I. F. Akyildiz, "Current status and directions of ieee 802.11 be, the future wi-fi 7," *IEEE access*, vol. 8, pp. 88 664–88 688, 2020.

[9] W. Zeng, M. A. S. Khalid, and S. Chowdhury, "In-vehicle networks outlook: Achievements and challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 1552–1571, 2016. DOI: 10.1109/COMST.2016.2521642.

[10] F. Geyer and G. Carle, "Network engineering for real-time networks: Comparison of automotive and aeronautic industries approaches," *IEEE Communications Magazine*, vol. 54, no. 2, pp. 106–112, 2016. DOI: 10.1109/MCOM.2016.7402269.

[11] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde, *The C# programming language*. Pearson Education, 2008.

[12] M. Sústrik *et al.*, "Zeromq," *Introduction Amy Brown and Greg Wilson*, p. 16, 2015.

[13] S. Popić, D. Pezer, B. Mrazovac, and N. Teslić, "Performance evaluation of using protocol buffers in the internet of things communication," in *2016 International Conference on Smart Systems and Technologies (SST)*, 2016, pp. 261–265. DOI: 10.1109/SST.2016.7765670.

[14] C. Currier, "Protocol buffers," in *Mobile Forensics–The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices*, Springer, 2022, pp. 223–260.

[15] Y. Chung, C. Park, and F. Harashima, "A position control differential drive wheeled mobile robot," *IEEE Transactions on Industrial Electronics*, vol. 48, no. 4, pp. 853–863, 2001.

[16] M. Crenganis, C. Biris, and C. Girjob, "Mechatronic design of a four-wheel drive mobile robot and differential steering," in *MATEC Web of Conferences*, EDP Sciences, vol. 343, 2021, p. 08 003.

[17] M. B. Kjærgaard, H. Blunck, T. Godsk, T. Toftkjær, D. L. Christensen, and K. Grønbæk, "Indoor positioning using gps revisited," in *Pervasive Computing: 8th International Conference, Pervasive 2010, Helsinki, Finland, May 17-20, 2010. Proceedings 8*, Springer, 2010, pp. 38–56.

[18] F. van Diggelen, "Indoor gps theory & implementation," in *2002 IEEE Position Location and Navigation Symposium (IEEE Cat. No. 02CH37284)*, IEEE, 2002, pp. 240–247.

[19] U. Wandinger, "Introduction to lidar," in *Lidar: range-resolved optical remote sensing of the atmosphere*, Springer, 2005, pp. 1–18.

[20] Y. Wu, Y. Wang, S. Zhang, and H. Ogai, "Deep 3d object detection networks using lidar data: A review," *IEEE Sensors Journal*, vol. 21, no. 2, pp. 1152–1171, 2020.

[21] G. Zamanakos, L. Tsochatzidis, A. Amanatiadis, and I. Pratikakis, "A comprehensive survey of lidar-based 3d object detection methods with deep learning for autonomous driving," *Computers & Graphics*, vol. 99, pp. 153–181, 2021, ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2021.07.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849321001321.

[22] M. Quigley, K. Conley, B. Gerkey, *et al.*, "Ros: An open-source robot operating system," in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.

[23] P. Estefo, J. Simmonds, R. Robbes, and J. Fabry, "The robot operating system: Package reuse and community dynamics," *Journal of Systems and Software*, vol. 151, pp. 226–242, 2019, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.02.024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219300342.

[24] P. Bouchier, "Embedded ros [ros topics]," *IEEE Robotics & Automation Magazine*, vol. 20, no. 2, pp. 17–19, 2013.

[25] A. Mili and F. Tchier, *Software testing: Concepts and operations*. John Wiley & Sons, 2015.

[26] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "Mock objects for testing java systems: Why and how developers use them, and how they evolve," *Empirical Software Engineering*, vol. 24, pp. 1461–1498, 2019.

[27] K. Takaya, T. Asai, V. Kroumov, and F. Smarandache, "Simulation environment for mobile robots testing using ros and gazebo," in *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*, IEEE, 2016, pp. 96–101.

[28] J. Platt and K. Ricks, "Comparative analysis of ros-unity3d and ros-gazebo for mobile ground robot simulation," *Journal of Intelligent & Robotic Systems*, vol. 106, no. 4, p. 80, 2022.

[29] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. Von Stryk, "Comprehensive simulation of quadrotor uavs using ros and gazebo," in *Simulation, Modeling, and Programming for Autonomous Robots: Third International Conference, SIMPAR 2012, Tsukuba, Japan, November 5-8, 2012. Proceedings 3*, Springer, 2012, pp. 400–411.

[30] H. R. Kam, S.-H. Lee, T. Park, and C.-H. Kim, "Rviz: A toolkit for real domain data visualization," *Telecommunication Systems*, vol. 60, pp. 337–345, 2015.