

Advanced Framework for Programming and Controlling Multi-Robot Systems in Self-Driving Labs

Integrating Behavior Trees and Reinforcement Learning for Automated Lab Procedures

Adshya Vasudavan Iyer Ibrahim Jad Masri

Robotics, Control and Automation

Master Thesis

Group 934, June 2023



Copyright © Aalborg University 2023

The report is written in \LaTeX via Overleaf. Illustrations has been made using GIMP, diagrams.net, eraser.io, code2flow.com, Weights and Biases, and Matplotlib.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.



Department of Electronic Systems
Aalborg University
es.aau.dk

AALBORG UNIVERSITY

STUDENT REPORT

Title:

Advanced Framework for Programming and Controlling Multi-Robot Systems in Self-Driving Labs

Theme:

Master Thesis

Education:

Robotics
Control and Automation

Project period:

September 2022 to June 2023

Project group:

Group 934

Participants:

Adshya Vasudavan Iyer
Ibrahim Jad Masri

Supervisor(s):

Simon Bøgh

Pages: 78

Appendix included: 92

Date of completion:

June 2, 2023

Abstract:

The increasing deployment of robots in factories demands cost-efficient and non-expert programming approaches. This project aims to develop an easy-to-use solution for controlling and creating robot tasks. The proposed approach utilizes intuitive and visual programming with Behavior Trees (BTs) and Reinforcement Learning (RL) for control, based on the principles of Skill-Based Systems (SBS). The focus is on applying this system in Material Acceleration Platforms (MAPs), specifically for Self-Driving Labs (SDLs). A Matrix Production System (MPS) with shuttles and manipulators served as a use case for validating the solution. Research showed challenges in automating lab procedures, such as task transfer and lab layout limitations, hinder the automation of lab work. The solution presented demonstrated capabilities of creating and executing BTs on the MPS and the RL agent successfully navigated obstacle-free environments but faced difficulties with multiple obstacles due to control and behavior tendencies. The system serves as a proof of concept but requires further improvements before being put to use. Overall, the novel combination of BT and RL for multi-robot systems in MAPs shows promise, in advancing the automation of lab tasks and material discovery.

Contents

Preface	iii
Acronyms	iv
1 Introduction	1
2 Problem Analysis	3
2.1 Platform for Self-driving Lab	3
2.2 Challenges in Self-Driving Labs	5
2.3 Related Works	7
2.4 Commercial Solutions	15
3 Methods	19
3.1 Behavior Trees	19
3.2 Learning-based Control	20
3.3 Simulation Software	26
4 Problem Formulation	29
4.1 Final Problem Statement	29
4.2 Objectives	29
5 Implementation	31
5.1 Design	31
5.2 Software Stack and Architecture	33
5.3 Device Manager	36
5.4 Behavior Tree	37
5.5 Skill Library	40
5.6 Reinforcement Learning Agent	45
6 Test and Results	55
6.1 Discrete Case	55
6.2 Continuous Case	61
6.3 Behavior Engine	64
6.4 Deploying the RL agent	66
7 Discussion	72
7.1 Evaluation of the Solution	72
7.2 Improvements	74
8 Conclusion	76
9 Future Work	77

Bibliography	79
A Appendix	83
A.1 What is a good user interface?	83
A.2 Finite State Machines	86
A.3 Exploration and Exploitation	89
A.4 Behavior Tree Implementations	90
A.5 XPlanar Library	90
A.6 Homogeneous Transformation Matrix	91

Preface

We would like to lead this report by expressing our heartfelt gratitude to all those who have contributed to the successful completion of this year-long master's thesis. The journey of this project has been as challenging as it has been giving, and in hindsight, we feel proud to see the vision we had in our minds for this project come to fruition. This research project would not have been possible without the invaluable support and guidance of the following individuals.

First and foremost, we would like to extend our deepest appreciation to our supervisor, Simon Bøgh¹, for his exceptional expertise and commitment throughout this thesis. His knowledge and passion for reinforcement learning have been influential in shaping the direction of this project. We value the encouragement, guidance, and positive attitude he has offered to us, which has made this thesis project both enjoyable and a great learning experience. We would furthermore like to thank Associate Professor Casper Steinmann² for providing us with valuable insight on lab experimental work procedures which has truly been helpful in analyzing the challenges posed when implementing a self-driving lab. This has been key for understanding how to enable lab workers to easily plan and execute lab experiments automatically. Finally, we would like to thank Assistant Professor Casper Schou³ for sharing his knowledge within skill-based systems and for inspiring us to create our own rendition of such a system.

It is our sincere hope that the research and findings presented in this report will contribute to the existing knowledge and promote future work within the field of skill-based systems and reinforcement learning. Videos of the solutions can be found on this [link](#).

¹<https://vbn.aau.dk/da/persons/118609>

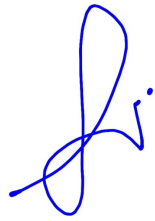
²<https://vbn.aau.dk/da/persons/casper-steinmann-steinmann>

³<https://vbn.aau.dk/da/persons/127366>

Acronyms

MAP	Material Acceleration Platform
SDL	Self-Driving Lab
SBS	Skill-Based Systems
BT	Behavior Tree
UI	User Interface
UX	User Experience
LiDAR	Light Detection and Ranging
SAC	Soft Actor-Critic
MDP	Markov Decision Process
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
FSM	Finite State Machine
PPO	Proximal Policy Optimization
ODE	Open Dynamics Engine
ML	Machine Learning
SLAM	Simultaneous Localization and Mapping
PMS	Process Management System
ROS	Robot Operating System
PMC	Planer Motor Controller
MPS	Matrix Production System

Signatures



Adshya Vasudavan Iyer



Ibrahim Jad Masri

Chapter 1

Introduction

In the recent years there has been a clear uptick in the deployment of robots in factories, as the industry today are including more robots in manufacturing processes than ever before. According to the World Robotics Report 2022, there has been a year-to-year increase of 33%, with 2021 having a number as high as 517,385 robots [1]. With the number of robots increasing the demand for new and easy alternatives for programming them arises, as the traditional methods for setting up robot tasks are both time consuming, costly, and requires expert knowledge [2]. While the idea of easy-programming interfaces among collaborative robots has been around for a while, it is still not a wide-spread practice in regard to industrial robots. Furthermore, programming multi-robot systems can be challenging even for expert programmers, let alone for non-technical users, which is why non-programming methods for robot task planning is interesting to explore to find solutions for this problem. This project presents a novel approach for easy programming of multi-robot systems using Behavior Trees (BTs) and Reinforcement Learning (RL) for task planning and control, respectively. The project will focus on easy programming of experimental procedures in Self-Driving Labs (SDLs), which keeps in tone with the current global interest in Material Acceleration Platforms (MAPs).

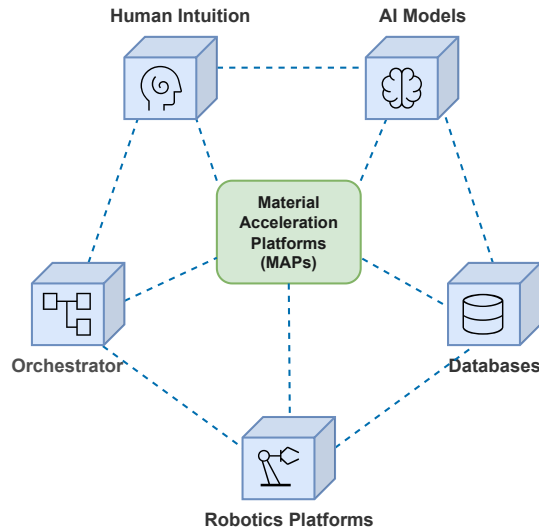


Figure 1.1: Illustration of the various modules combined which MAPs consist of. This figure is adapted from [3].

In the context of this project, MAPs refer to self-driving labs for material discovery. These labs commonly include robots and AI systems to help speed up experimental procedures, resulting in 10-100x faster discovery rates [4]. Fig. 1.1 shows the different

aspects and modules which all combined creates MAPs in a closed-loop approach. The AI models define and suggest the experiments which will be executed by the robotic platform. The orchestrator manages the data gathered during execution and updates the AI model. The constraints of the system along with database management are handled by the researchers. The researchers also use their prior knowledge as input to the AI models [3].

Currently, Denmark is investing a lot in research in MAPs with the newly founded pioneer center CaPeX. CaPeX is a project, which has eight universities (five national and three international) collaborating, working on the research and development of new materials [5]. The project presented in this report can be considered as pre-work for the upcoming research projects on MAPs at Aalborg University, yielding some insight within the area of automating lab processes. Thus, lay the grounds for other future research within MAPs, in regards to the robotics aspect.

The goal of this project is to develop an advanced framework that combines behaviour trees and reinforcement learning techniques to simplify robotic task planning and programming, for enhancing the efficiency of experimental processes. The focus will be on a task-level programming approach which will empower lab workers to efficiently program and control multi-robot systems in SDLs. This leads to the following initial problem formulation:

How can a multi-robot system be programmed and controlled using the capabilities of behavior trees and reinforcement learning techniques in order to automate lab procedures?

Chapter 2

Problem Analysis

This chapter introduces the hardware used in this project and the challenges of SDLs. Furthermore, it shows the current trend in robotic systems for task planning and control, leading up to the current state-of-the-art systems used today both in research and commercially available solutions.

2.1 Platform for Self-driving Lab

As a precursor for the solution presented in this project a lab setup is needed. For this, the Matrix Production System (MPS)[6] at AAU is used to simulate an SDL. The setup will be used for testing the implementation of the skill-based system along with the reinforcement learning control aspect.

Matrix Production System

The self-driving lab consists of an ACOPOS 6D planar motor system as well as 6 KUKA robots mounted on a tabletop. The system can be seen in Fig. 2.1 below.

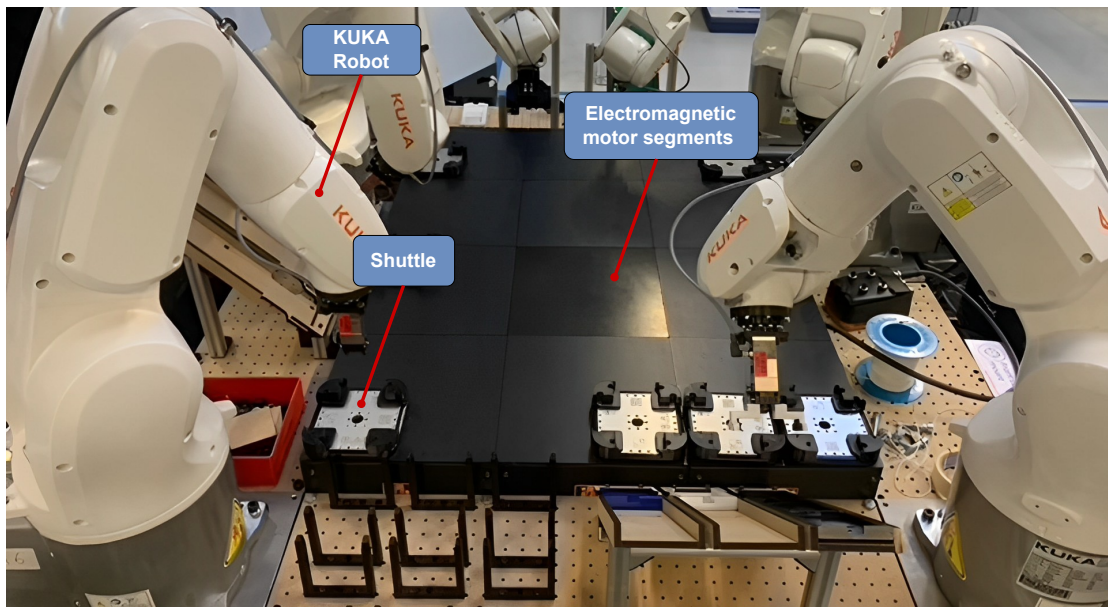


Figure 2.1: The MPS setup located in the AAU Smart Production Lab consists of a B&R planar motor system called ACOPOS 6D and six KUKA robots [6].

The ACOPOS 6D system is developed by B&R Automation in collaboration with Planar Motor. The system consists of magnetically levitated shuttles that can be moved in 6 degree-of-freedom on top of electromagnetic motor segments. The shuttles come in a variety of sizes and can carry different payloads. Each shuttle has a unique ID making

it possible to keep track of them. In the lab, the smallest models are used with a size of $118 \times 118 \text{ mm}$. The shuttles are capable of moving precisely with less than 5 microns in repeatability with a maximum speed of 2 m/s and maximum acceleration of 20 m/s^2 . The motor segments are identical in size and modular in configuration allowing the layout of the system to be changed with ease [7]. As can be seen in Fig. 2.1, the ACOPOS system at AAU, consist of 7 shuttles that can move around on a platform consisting of 3 by 4 motor segments. Each motor segment has its coordinate system, but when daisy-chained, a global reference frame is used instead.

This flexibility and scalability of the system make it a good choice for a self-driving lab. Furthermore, as the system is magnetically levitated, the motion is contactless making it maintenance-free. The shuttles are driven by a central controller given a position, maximum velocity, maximum acceleration, and end velocity. The ACOPOS system is controlled using a PLC configured to work with the Planar Motor system. Alternatively, it is possible to use the Planar Motor API by connecting directly to the Planer Motor Controller (PMC) [8].

There exist other solutions similar to the ACOPOS system, such as XPlanar which is developed by Beckhoff Automation and offers similar system capabilities [9].

The KUKA robots are of two different types consisting of 4 KUKA KR3 R540 and 2 KUKA KR4 R600. Each robot has its controller and can be interfaced with from a central PLC which all the different robot controllers are connected to. Alternatively, each robot can be controlled individually using a TCP/IP server running on each robot controller [10].

In this project, it is desired to make it easy to program robot tasks, control different types of robots as well as use reinforcement learning to control the ACOPOS 6D system to do local path planning and collision avoidance.

2.2 Challenges in Self-Driving Labs

When researching the difficulties that may involve in developing a SDL, it uncovered some relevant issues, that should be taken into consideration. The first is the lab consisting of a heterogeneous system. Heterogeneous systems consist of multiple types of processing units, which poses challenges due to each processing unit has its protocol and API to communicate with it [11]. Another major challenge, relevant to the process of automating the experimental tasks in the lab, is the translation of manually performed tasks which are complex. The translation of an experimental task designed to be performed by a human is not directly transferable to a robot [12]. In the context of automating the task, the manner of how the task is performed may differ, with both the setup and the motion planning, as there may be a more optimal way of doing so. Therefore a lot of thought should go into the design and planning of the experiments. Related to this problem, is the issue of handling lab equipment, such as assembling and disassembling equipment as well as utilizing the equipment for tasks such as dispensing materials. According to the [12], automating the procedure of dispensing liquids is considered to be easier than dispensing solid substances (powder), especially in the matter of dispensing really small quantities of powder substances. This can be explained, as working with dispensing larger amounts of powders can resemble more fluid-like flow, while working with smaller quantities, every single solid particle is significant, and requires a high level of accuracy when dispensing. This makes it challenging to automate many of the needed experimental tasks.

To further investigate the validity of the concerns and challenges mentioned above, the project group was granted a lab tour by Casper Steinmann¹, Associate Professor in Computational chemistry at AAU. The tour gave some valuable insight into the processes that go into performing experiments as well as improvements that can be made to these. The procedure for performing experiments can be divided into 4 different steps; the first step is preparation, which takes place before entering the lab. The preparation can take days and consist of designing the experiments, writing down the equipment needed, and making risk assessments according to the safety protocols for the lab procedure. The second step is to prepare all equipment and substances when entering the lab before starting the experiment, this involves measuring out the quantities needed by hand. Once these steps are completed, one can move on to the third step which is to perform the series of desired experiments, followed by the final step of analyzing the results.

The information gathered from the lab suggests that the current state of today's labs concurs with the concerns and assumptions stated in the above section. Some of the key points are the concerns around dispensing materials, as this is currently done manually today as seen in Fig. 2.2, which is considered to be a very difficult task to automate. This is due to the fact that the amount needed is too marginal, measured in milligrams down to micrograms, yet each particle has a significant impact on the experiments and results.

¹<https://vbn.aau.dk/da/persons/casper-steinmann-steinmann>

Another point in regards to dispensing is the lab has a large variety of containers in all shapes and sizes, as seen in Fig. 2.2 (left) making it too complex for the robot to be able to aid with the preparation of the experiments. Therefore it could be concluded that the more feasible area to introduce automation in the lab is in conducting the experiments, and not in the design and preparation of it, as this requires human intervention.



Figure 2.2: Right figure shows Casper Steinmann, Associate Professor in Computational Chemistry at AAU, demonstrating how substances are still measured manually today, during the lab tour at the Department of Chemistry and Bioscience. The left figure, has Casper S. showing all the varieties of containers, that are found in the lab.

In an attempt to break down the task of automating experiments, some of the main issues that surface is the current layout and equipment of the lab designed for humans operating in the workspace is not adaptable for robot integration. An example of this is the fume hoods, seen in Fig. 2.3, which are too small to have a robot operating inside it, furthermore, the fume hoods tend to become cluttered and cramped with equipment and materials, which is not ideal for a robot setup. This has led to the conclusion that the self-driving lab must have a less conventional setup with appropriate measures concerning the robot workspace. Furthermore, it is determined that the most feasible way to accelerate experimental procedures and provide results faster would be to aim for multiple experiments running in parallel on the platform. With this in mind the MPS platform presented earlier is well argued for.



Figure 2.3: Picture of one of the fume hoods found in the Chemistry and Bioscience lab, which shows how it would not be suitable having a robot operating inside of it, due to the robot's size and all the equipment inside of the hood.

2.3 Related Works

This section will present the findings of the research made on related projects, solutions, and the development within the topics affiliated with this project. The section is divided into three parts, research on MAPs, RL followed by research and current solutions for skill-based systems.

2.3.1 Material Acceleration Platforms

To plan experiments in SDLs, Roch et al. (2020) [13] present a software package called ChemOS. It is developed as a versatile agnostic software that supports the high-level experimental scheduling of both fully autonomous workflows as well as experiments with human intervention involved. The system uses Machine Learning (ML) to deduce which experiments to proceed with based on feedback from previously conducted experiments.

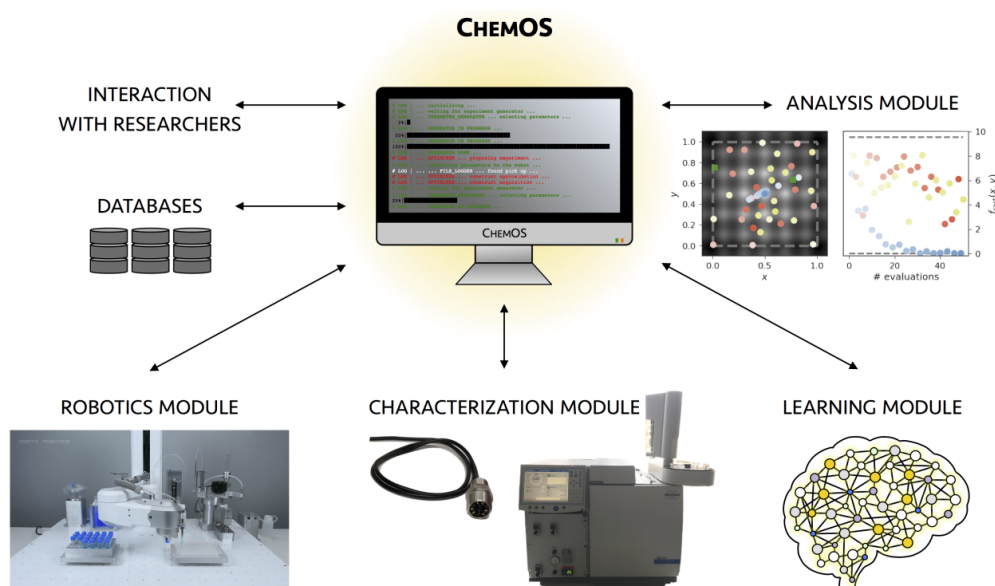


Figure 2.4: The different modules of ChemOS showing the interface and the module names [13].

ChemOS is comprised of different modules, shown in Fig. 2.4, that are commonly needed in autonomous labs. The first module is the user interface for interacting with the researchers making it possible to create new requests for experiments. This module also comes with a ChatBot which gives a predetermined response based on the category of the message received. The second module is database handling and management which is used to save and retrieve feedback, parameters, requests, and results in different databases. The third module is the robotics module which acts as a bridge between ChemOS and the hardware used in the experimentation. The fourth module is the characterization module which takes high-level instructions and translates them to low-level elementary machine-oriented commands. The fifth module is the analysis module which is used to visualize data to analyze it. Lastly, the learning module is the key to creating autonomous experiments and is used for parameter space search and decision-making. Even though ChemOS provides a robotics module, it has limitations as it does not include any hardware drivers or communication protocols to ease the integration of robotics in self-driving labs.

Mobile manipulators are proven beneficial for automating chemical procedures. Burger et al. (2020) [14] present a concept of a mobile chemist which is used to perform tasks similar to a human lab worker. Here, Burger et al. used a platform known as KUKA Mobile Robotics (KMR) with a KUKA iiwa manipulator as shown in Fig. 2.5. Simultaneous Localization and Mapping (SLAM) algorithms were used to map the lab and allow the robot to navigate to target stations. The researchers mention that the positional capabilities of the system were ± 10 mm making it possible to navigate to different stations but limiting fine manipulation capabilities. Therefore, the team added touch sensors to make it more precise.



Figure 2.5: The KMR iiwa platform is used to automate chemical experimentation by going to different pre-defined locations, shown in orange on the map. The locations correspond to different stations used for dispensing, loading, storing, etc.[14]

The mobile robot moves to pre-defined locations on the map to get to the stations to perform the scheduled tasks. The stations are controlled by a Process Management System (PMS) over TCP/IP and RS-232 communication protocols. This made it possible to schedule and execute tasks in parallel where the different machines are working simultaneously enabling the system to process more batches. Similar to ChemOS, different search algorithms were used to learn which tasks to perform next in order to achieve the desired material properties. It is worth noting that the researchers spent approximately 2 years developing the solution most of which involved working on protocols and software automation.

It can be seen that both ChemOS and the mobile chemist have similar needs. Both systems have to communicate or bridge hardware and integrate it into the system and plan the experiments along with optimizing the search space and storing the results, either physically or digitally.

2.3.2 Reinforcement Learning

Although there are many different path-planning algorithms that can be used for managing the shuttles in the ACOPOS system, many of these traditional algorithms cannot adapt to changes in the environment, which can make it hard to move around in an unknown environment, while avoiding obstacles.

In recent years, a lot of research revolving around the use of reinforcement learning for control and obstacle avoidance has been made, for applications such as self-driving cars, trajectory planning for robot manipulators, and path planning of mobile robots, to name a few. In Choi et al. (2021) [15], an approach for dynamic obstacle avoidance using reinforcement learning for mobile robots is presented. The research combines Deep Reinforcement Learning (DRL) and the integration of path planning, for optimal control. What is of interest in this study is how the collision is defined as an RL problem. To uncover this it is relevant to look into how the states, actions, and rewards are defined.

$$s_t = [s_t^{lidar}, s_t^{goal}, s_t^{speed}] \quad (2.1)$$

According to [15] the state is defined as seen in Eq. (2.1) which also corresponds to the observations given to the mobile robot (agent). The observation is the data received from a Light Detection and Ranging (LiDAR) scanner, which consist of the distance to its surroundings and potential obstacles, the relative distance to the goal position in x and y, as well as the forward- and rotational velocity v and ω , respectively.

$$a_t = [v, \omega] \quad (2.2)$$

The agent action is described as continuous behavior as it was desired to have more smooth control and is considered more advantageous when dealing with obstacle avoidance. The actions, Eq. (2.2), are based on the two components the mobile robot receives at each time step, one being the forward velocity, v , and the second rotational velocity, ω .

$$R = R_g + R_c + R_\omega \quad (2.3)$$

The reward function, Eq. (2.3), given in [15] is based on a sparse reward system and gives an intuition of how to design a reward function for the shuttles. The rewards consist R_g which is the completion reward of +10 when reaching the goal position combined with a smaller positive reward when being in the vicinity of the goal. R_c is the -10 penalty for colliding with obstacles, which also causes the episode to terminate. R_ω is the penalty that the mobile robot receives when exceeding the rotational velocity above a certain set threshold.

The RL agent is based on the Soft Actor-Critic (SAC) and the actor-network can be seen below in 2.6. The actor-network is a multi-input CNN that takes the observation data

(LiDAR data, distance to goal, and velocity) as input and outputs the probable action as velocities.

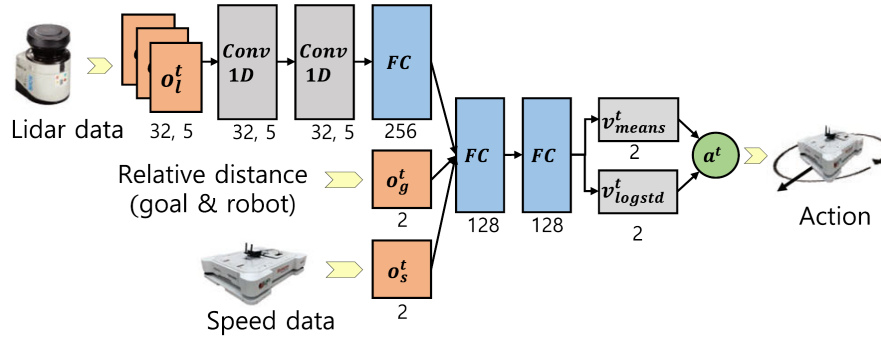


Figure 2.6: The diagram shows an overview of the actor-network of collision avoidance learning using SAC. The network takes 3 inputs, LiDAR data, distance to the goal, and velocity. It outputs an action consisting of velocities [15].

2.3.3 Skill-Based Systems

Important research that has laid the groundwork for Skill-Based Systems (SBS) is the project presented by Rath Pedersen et al. (2015) [16], which has inspired many derivative works in recent years. The paper describes how to use robotic skills for task planning. According to the research, any task can be defined as a set of skills. A skill can be a sub-task such as 'pick up', 'place', or 'assemble'. A robot skill is described as an object-oriented ability, in parameterized form, which can be composed into tasks. The skill construction itself is considered generic but can be configured to be task-dependent using the parameters.

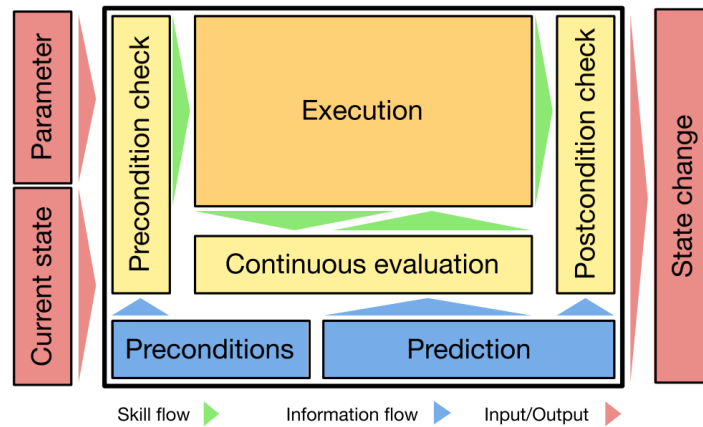


Figure 2.7: Diagram showing the structure of a robot skill with all its sub-elements as presented in Rath Pedersen et al. [16]

Fig. 2.7 shows how a robot skill is defined. A skill takes an input parameter such as object location, and the current state of the robot. Before the skill can be executed the system will check whether the pre-condition for the given skill is met. The performance

of the skill is constantly evaluated during execution and is validated by a post-condition check, once the skill is completed before the system will move on to update the states. The system uses state machines to transition between states.

The general architecture of the system is shown in Fig. 2.8, which consists of three abstract layers. The bottom layer contains the device primitives, which are the low-level control enabling the control of the hardware, gaining data, and communicating with the actuators, sensors, and other devices. A device primitive exposes the hardware capabilities to the skill layer, which can be combined into a robot skill.

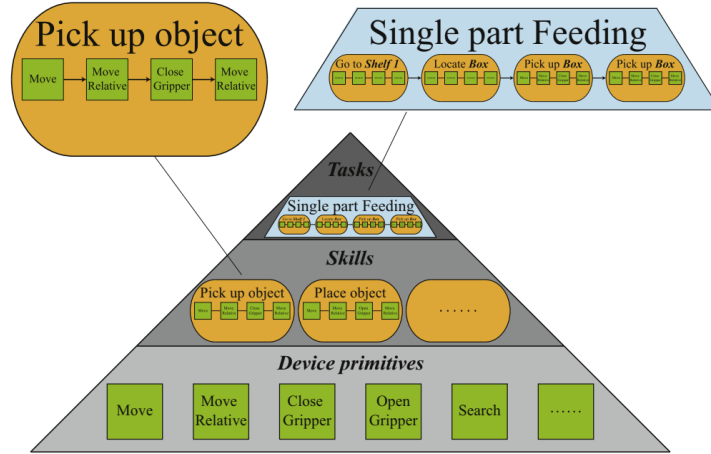


Figure 2.8: The figure shows the three abstract layers that the SBS commonly consist of as presented in [16]. At the bottom are the device primitives, the second layer is the skill layer, and above those two is the task layer.

The skill layer is the layer relevant for being able to configure tasks and reconfigure them as desired for different production tasks. While the functionality of the robot task is still implemented manually today, having robot skills can ease the process of having to program the robot system and task from scratch. The task level shows the sequences of skills created and gives an overview of the general task setup for the system, but it is the skills that determine the functionality and the behavior of the robot, which is why the skills are also the system component that is responsible for updating the states of the robot and the world (environment). This research is considered a pivotal point for automating robot programming, making it easy for both integrators and employees that play a direct role in the use of production lines e.g. operators to be able to plan new robot tasks or setups, with existing hardware. This alleviates the redundancy and time spent on robot task planning for customization needs in the industry. Some of the authors connected to the research, had continued to develop the above-mentioned system, publishing the Schou et al. (2018) [17].

Andersen et al. (2017) [18] have used a system similar to the mobile chemist [14], called Little Helper, with a focus on Industry 4.0 by integrating the skill-based system. The platform used a MiR100 base with a UR5 as the manipulator. Unlike the mobile chemist, the Little Helper uses skills to create and execute tasks along with a device manager making its deployment quicker compared to the mobile chemist.

Mayr et al. (2021) [19] take the concept of SBS one step further. The research uses reinforcement learning to learn the optimal parameters needed as input for a task. Unlike [16], this work uses Behavior Trees (BTs) to represent a policy that determines the behavior of the robot at a task instead of state machines. The system proposed in this paper is divided into 3 main components, behavior trees based on movement skills, policy parameter optimization, and domain randomization.

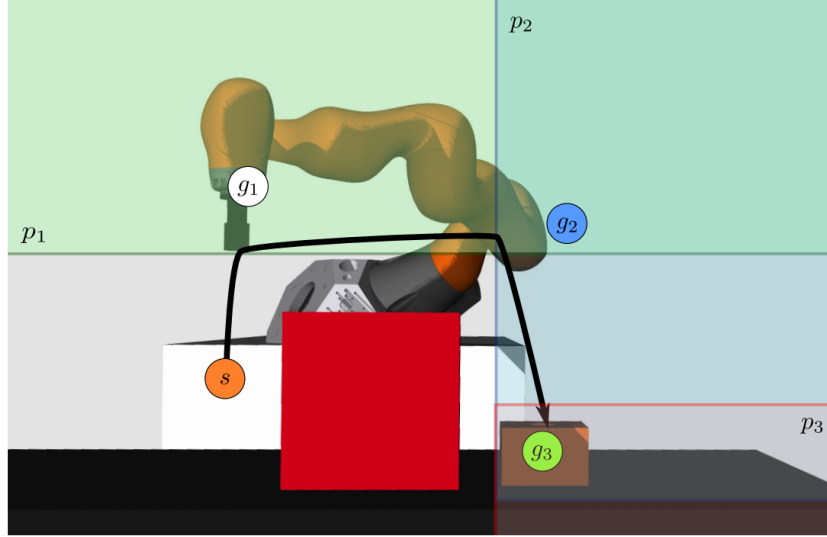


Figure 2.9: Visual representation of a peg-in-hole task consisting of three goals in different areas. The goal of the robot is selected based on which area the end-effector is in [19].

Fig. 2.9 shows a peg-in-hole task with different goals to avoid collision with the red square, representing an engine block, as well as to perform the peg insertion. The task has been described in terms of a behavior tree, where the robot starts at point s and has to go to target g_1 if the end-effector's position is in area p_1 . Otherwise, the robot has to go to g_2 if its end-effector's position is in area p_2 . Finally, the main goal the robot had to reach is g_3 in area p_3 . The reactivity of the behavior tree allows for periodically checking the position of the end-effector to determine which target goal the robot must move to. This differs from the skill-based system where continuous evaluation is done in the skill itself. Here, the conditions are not a part of the skill, but rather their own component.

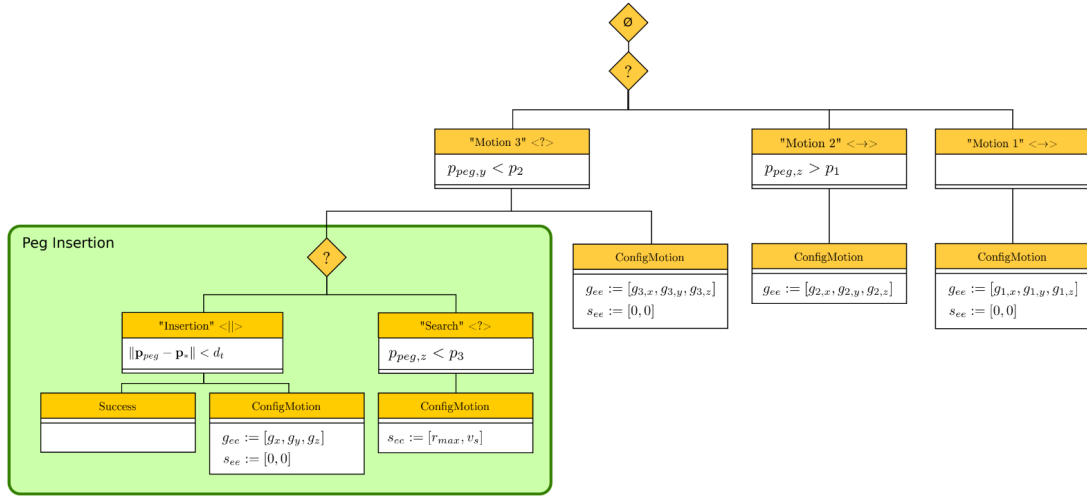


Figure 2.10: Diagram showing the behavior tree structure for the peg-in-hole task represented in [19].

As it can be seen in Fig. 2.10, the tree has three condition nodes to check the end-effector position. If the position is close to the obstacle, the goal for the robot will be a point that is far from the object. However, when the robot is close to the hole, it starts the peg insertion task (green box). In the figure, changing the tasks of peg insertion to another task can be done by replacing the peg insertion subtree with another one. This demonstrates the capabilities of the BT approach to task planning.

2.4 Commercial Solutions

The following section investigates the commercially available solutions, covering their capabilities. This section is used to draw inspiration for designing the solution presented in this project.

2.4.1 ForgeOS

ForgeOS is a commercially available solution that is developed by the US-based company Ready Robotics. The solution combines easy robot programming and task planning on a common platform that supports robots from various robot manufacturers, sensors, and cameras. ForgeOS consist of flowchart-based programming that comes along with a teach pendant, with the concept of plug-and-play. It enables the user to easily program robot tasks with the approach of visual programming (block-programming), without the difficulty of integrating hardware into the system manually. The system is based on state machines and is the first universal operating system for industrial robots. It is built on Qt and C++ making it available across platforms. The main components of the system are the Task Canvas where the robot tasks can be created, Device Control which enables the user to seamlessly control the devices connected to the system in real-time, and the Device Configuration where the user can add and configure the desired robot and additional hardware to their work-cell [20].



Figure 2.11: Image showing the user interface of ForgeOS developed by Ready Robotics, which uses finite-state-machines for programming of robots and supports a large variety of robot manufacturers [20]

ForgeOS comes with different blocks that can be dragged and dropped. These blocks consist of high- and low level skills such as open and close gripper (high), and read and write IO signals for a specific device (low) [21].

2.4.2 MoveIt Studio

MoveIt Studio is developed by PickNik Robotics, another US-based company. MoveIt studio, shown in Fig. 2.12, is a software framework that enables the user to program and test robot hardware remotely and to easily perform task planning and motion planning. The software targets both non-experts and experts and supports easy debugging and inspections of trajectories for recovery. One of the advantages of MoveIt Studio is its integration with Robot Operating System (ROS). This enables the user to design robot behaviors and use many of ROS and MoveIt capabilities with an easy-to-use graphical interface based on behavior trees. Since the software supports ROS, it means many of the features supported by ROS are also supported such as navigation for mobile robots and manipulator trajectory planning. Additionally, the hardware supported by ROS can also be used and configured to work with MoveIt Studio. The software also supports remote access, to program and command the robot. Furthermore, it supports debugging and monitoring the state of the behavior tree as well as other monitoring systems metrics such as CPU load, network usage, and RAM usage [22].

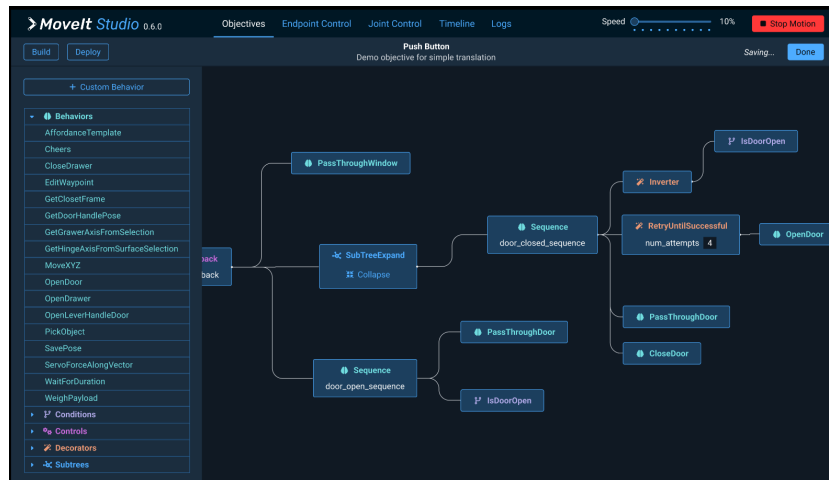


Figure 2.12: The MoveIt Studio user interface, developed by Picknik. MoveIt Studio uses behavior trees where the nodes can be dragged and dropped from a side panel [22]

MoveIt Studio is based on an open-source behavior tree implementation called BehaviorTree.CPP² written in C++. This implementation is designed in a way that makes it easy to create new nodes which is done by inheriting from base classes depending on the needed functionality. These nodes have to be registered in a behavior tree factory which then makes them available to be used. The execution nodes in this behavior tree

²<https://www.behaviortree.dev/>

implementation are asynchronous allowing concurrent execution of nodes. The different variables have to be defined on a blackboard. A blackboard is a dictionary of keys and values, where the user is responsible for creating the blackboard and adding the required keys and values to use them later as an input to or output from other nodes. This is done in the same manner in MoveIt Studio. Furthermore, the execution nodes have ports, which are a way to make sure that variable types are checked along with whether it is an input, output, or inout variable.

2.4.3 RiFLEX

Similar to the above-mentioned solution, RiACT, a Danish-based startup, offers a common platform called RiFLEX to program robot tasks that supports many robot manufacturers, in the form of a skill-based system. They focus on visual programming that enables the user with no programming skills to be able to create new tasks. Fig. 2.13 shows the user interface of their software with a component sidebar as well as different skills to program the robot. Similar to MoveIt Studio, RiFLEX is also integrated with ROS leveraging its capabilities as a robotics middleware. The system core functions, referred to by the company as the three c's are; connect, configure and coordinate. The capabilities of connecting robots and other hardware in the work cell, configuring the connected hardware to create tasks with the help of generic skills, and coordinating the system so it seamlessly integrates into the production line. Unlike ForgeOS, RiFLEX uses extended behavior trees to construct tasks consisting of skills. The solution comes with an HMI with their software which allows the user to connect it to the hardware setup in the production line. The system also supports simulation with a digital twin setup for learning, planning, and knowledge sharing [23].

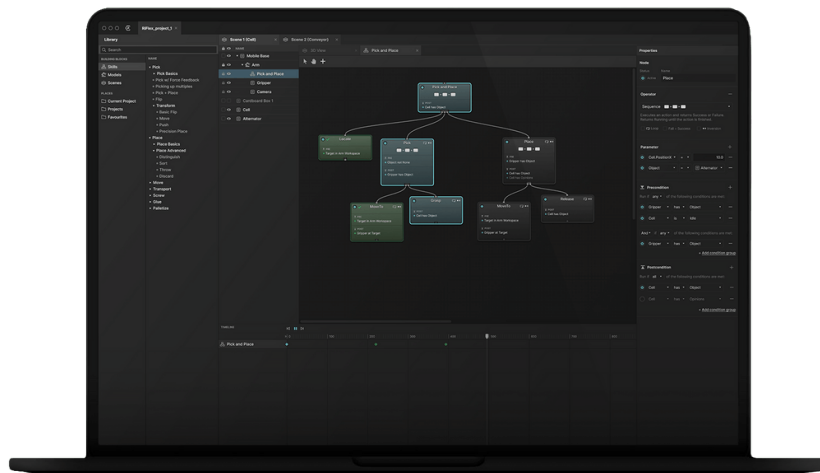


Figure 2.13: The user interface for the RiFLEX software developed by RiACT. RiFLEX is based on behavior trees, providing easy programming of robots [23].

RiFLEX software is based on a project called Skiros2³, which is written in Python. In this implementation, the skill designer creates a new skill by extending a base class and defining different pre-, hold- and post-conditions for the skill. Unlike BehaviorTree.CPP, the conditions are not separate nodes by themselves but are integrated into the skill itself. Similar to BehaviorTree.CPP, the execution of the nodes is done asynchronously.

2.4.4 Summary

It can be seen that all the commercial solutions have a well-designed user interface allowing the user to create tasks easily. Additionally, hardware configuration seems to be an important aspect of the solutions where both RiFLEX and MoveIt Studio leverage the power of ROS to integrate more hardware into their solutions. On the other hand, ForgeOS uses other methods to configure hardware. RiFLEX and MoveIt Studio use different implementations of behavior trees with similar core capabilities. RiFLEX uses only high-level skills based on a strictly defined structure, while MoveIt Studio and ForgeOS give more freedom in how to design the task by providing both high- and low level nodes.

³<https://github.com/RVMI/skiros2>

Chapter 3

Methods

This chapter will introduce the reader to the fundamental theory and methods considered when implementing the solution presented in the following chapters. The first section describes the idea behind behavior trees followed by the learning-based control section which dives into the Markov Decision Process (MDP) which forms the basis for RL, used for controlling the shuttles in the self-driving lab.

3.1 Behavior Trees

Behavior trees are a powerful tool that can be used to describe the complex behavior of agents much more easily. BTs were originally used in game development for designing the behavior of non-playable characters but have gained a wide interest in them within robotics. BTs are composed of nodes connected by edges forming a tree structure. The tree starts with a root node that is connected to control nodes, ending with the leaf nodes which are known as execution nodes. Control nodes are made of 4 categories of nodes, sequence, fallback, parallel, and decorators. The execution nodes are made of 2 categories, action, and condition nodes. The execution of a BT starts when a tick is sent to the root node, which routes this tick to its children until they reach the execution nodes. Each node returns a state corresponding to success, failure, or running, depending on the state of execution. This state is then propagated back to the parent node. The tick is sent at a specific interval which makes the tree reactive to changes [24]. The behavior of the flow control nodes is explained below:

Sequence:

This node ticks its children in a specific order. If one of the children fails, the sequence is terminated and the sequence node also fails. This node needs all children to succeed to execute successfully [24].

Fallback:

This node ticks its children in a specific order. Unlike the sequence, if a child fails, the fallback ticks the next child. If all children fail, then this node fails too. This node needs only one child to succeed in order to execute successfully [24].

Parallel:

This node ticks all of its children at once. It is similar to a sequence node in which all children have to succeed for the parallel node to return success. Otherwise, it returns failure if one of its children fails [24].

Decorators:

These nodes have one single child and they behave according to a user-defined rule. An example of this is an inverter node which could be implemented in a way such that if a node returns failure, it inverts it and returns success instead. The

inverter node can be useful when used with condition nodes [24].

BTs are easier to understand based on their graphical representation. They are also composable where the user can plug an entire tree as a sub-branch of another tree. Furthermore, BTs ensures reusability where different behaviors can be made into sub-trees to be used with another tree [24]. An alternative for BTs is Finite State Machines (FSMs) which are described in Appendix A.2, where a comparison of the two is also presented.

3.2 Learning-based Control

RL is built upon the principles of MDP. This section will briefly explain the fundamentals of MDPs, followed by the theory relevant to understanding RL, and finally ending with Proximal Policy Optimization (PPO), which will be used in this project.

3.2.1 Markov Decision Process

MDP is a mathematical abstraction commonly used to describe an RL problem. It is used to describe the relationship between a decision maker, referred to as an agent, and its environment through states and actions. MDPs describe the sequential decision-making in an environment assumed to be fully observable. The importance of MDPs for RL is that they can be used to define states and their transition probabilities. Another relevant principle is the Markov property, also known as the memory-less property of a stochastic process, which determines that the future states only depend on the current state. In other words, any future state only depends on its prior state [25].

MDP principles

MDPs [25] are described as 5-tuple (S, A, T_a, R_a, γ) :

- S consists of a finite set of possible states in the given environment.
- A is the finite set of possible actions in a state.
- T_a represents the state transition matrix, which is the probability of transition from state S to state S' for a given action a at a time t .
- R_a is the reward obtained after transition from state S to state S' for an action a .
- γ is considered the discount factor that determines how the current and future rewards should be weighted, $\gamma \in [0, 1]$.

3.2.2 Reinforcement Learning

To dive deeper into the understanding of MDP and RL certain terminology must be presented. As mentioned these methodologies consist of agents, states, actions, and rewards, which will be explained further in the following sections.

Agent and Environment

The agent is an entity that will interact with the environment and has to achieve some desired goal. The RL agent is inserted into the environment and will learn how to perform the specified task optimally by observing the environment. The agent will achieve its goal by trial and error and will be rewarded based on behavior and performance, which it will try and optimize through actions. In other words, the agent will try and maximize the sum of rewards to achieve the optimal behavior for the given task. As for the environment, it can be based on a certain given model, so the environment is known, or the environment can be unknown. The relationship between the agent and the environment is illustrated in Fig. 3.1 below [26].

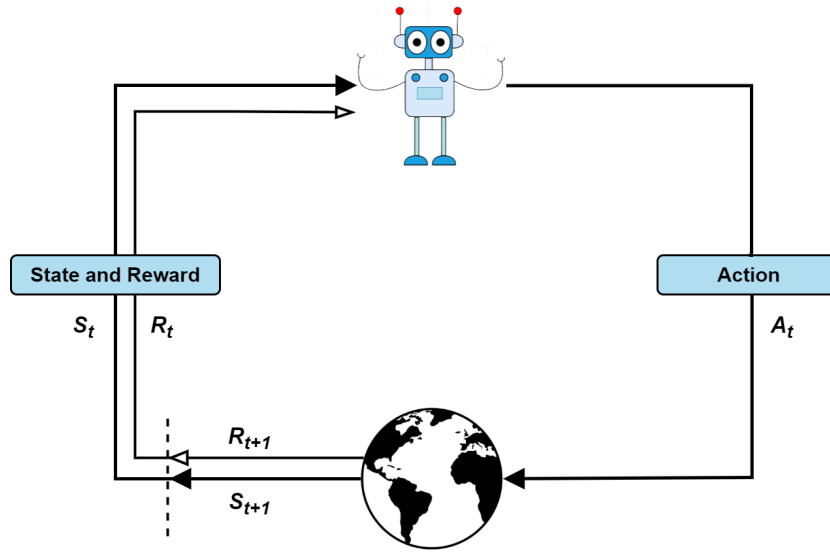


Figure 3.1: Illustration of the relationship between the agent (robot) and the environment, which can be translated into states, actions, and rewards.

As shown in the figure, the interaction between the agent and the environment can be described as a discrete case. At every time step t the agent receives an observation from the environment, based on this observation it performs an action a . Once the agent performs the action, the state of the environment will be updated and the agent will again receive a new observation in the form of the state and reward, based on the action taken.

Rewards

As mentioned above the reward can be a part of the observation that the agent receives once an action is performed. The rewards tell how well the agent is performing a certain task and is used by the agent to determine its future action during training. In other words, the agent learns how to perform well and maximize the accumulative sum of rewards commonly referred to as the return, through rewards. The rewards can be positive or negative (penalty) and are received at each time step. The return is denoted as G_t and can be derived from the following Eq. (3.1) for each time step:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots \quad (3.1)$$

To maximize the return another term called value function, V_s , must be introduced. The value function is used to determine potential future rewards in the current state. The value of a state is seen as the sum of the probability of reaching that state times the reward for that state. More precisely, the value of a given state is merely the sum of all the next states' probability multiplied by the reward for reaching that state [27]. The value function, V_s , can be further defined as the following equation:

$$v(s) = \mathbb{E}[G_t | S_t = s] \quad (3.2)$$

$$= \mathbb{E}[R_{t+1} + R_{t+2} + R_{t+3} + \dots | S_t = s] \quad (3.3)$$

Eq. (3.2) defines the value at state s at time t determined by the expected return in that given state s .

The return and the value term should not be considered supervised feedback, as it does not decide what action the agent must take. It is only used to inform the agent of the profitability of each state and action. How the agent will make use of the received information is based on the idea of exploration and exploitation [27]. The explanation of the exploration and exploitation can be found in Appendix A.3.

States and Actions

The state and the actions depend on each other. The state can be considered as the observation that includes the information on the current status of the environment, which is received by the agent. The actions determine what the agent can do in the current state. When the agent performs an action and the environment will receive the action taken and update its state and the agent will get a new observation, with a new set of possible actions that can be taken in the new state. The agent may seek more immediate rewards, which will lead to it choosing actions with the highest reward in the current state. But this strategy may have a negative impact on the rewards long-term which can prevent it in achieving the global maximum of the rewards. To avoid this problem it is important to be able to define the agent's behavior, which is why policies are essential for reinforcement learning [26].

Policy

While value functions determine the value of a given state, meaning how good it is to be in a certain state, the policy determines the probability of reaching that state. The value function is therefore tightly correlated to the policy that the agent follows. The policy can be seen as the strategy that the agent uses to reach its goal, and the actions taken can be described as the function of the agent's state and the environment [28].

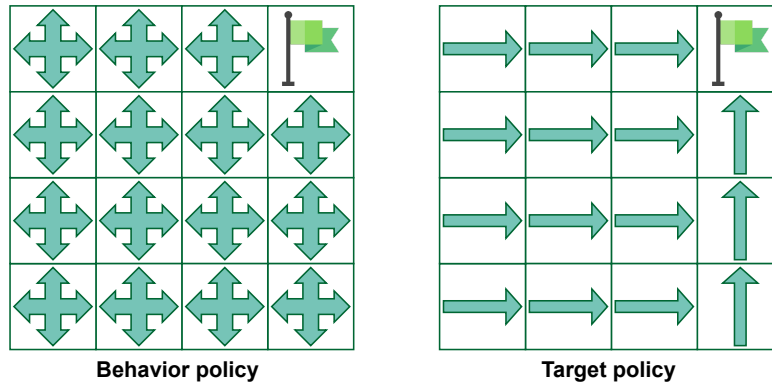


Figure 3.2: The figures illustrate behavior policy and target policy to explain the principle of on - and off policy. With On-policy the behavior - and target policy are the same, while with off-policy the agent follows a behavior policy in order to obtain an optimal policy (target policy).

The policy denoted as π decides on the agent's behavior by either following an on-policy or off-policy approach. A simple way of understanding the principles of on-policy and off-policy is to get familiarized with these two terms; behavior policy and target policy. The behavior policy is the policy that determines the agent's actions in each state while the target policy is the policy that the agent uses to learn from the received rewards based on the actions taken. This policy is used to update the Q-value (overall expected rewards). When an algorithm is on-policy the behavior policy and the target policy are the same, which means that the agent evaluates and updates the same policy it uses to optimize it. With off-policy algorithms, the agent uses a behavior policy to learn to explore the environment through actions but may be independent of the target policy which is being learned and optimized [28][26]. The two terms; behavior - and target policy, are illustrated in Fig. 3.2 for a more intuitive understanding [26].

3.2.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is one of the more commonly used algorithms for RL, which can be used for both discrete and continuous actions. The PPO algorithm is based on actor-critic architecture, which uses two separate models. One is the actor model which learns the actions to take based on the current state of the environment, and the second is the critic model which learns to evaluate the quality of the chosen actions. The PPO algorithm uses a novel approach to updating the actor policy, which has an improvement in the stability of the training. PPO is mainly used for its ability to constrain the policy update, such that deviation between the new and the old policy is not too large [29] [30].

The diagram shows the equation $L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right]$. Annotations include:

- Ratio function**: A red line pointing to the $r_t(\theta)$ term, which is enclosed in a red box.
- Unclipped term**: A blue line pointing to the $r_t(\theta) \hat{A}_t$ product, which is enclosed in a blue box.
- Clipped Objective**: A green line pointing to the $\text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t$ product, which is enclosed in a green box.

Figure 3.3: The Surrogate objective function

The equation given in Fig. 3.3 shows how PPO uses a surrogate objective function to constrain the policy update. The surrogate objective consists of two terms, the probability ratio, and the clipped objective.

$$r_t(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)} \quad (3.4)$$

The probability can be defined as Eq. (3.4). The probability ratio term determines how much the new policy deviates from the old policy, by looking at the ratio between the probability of the action taken by the agent under the new policy divided by the same action under the old policy [29, 30].

$$\frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)} \hat{A}_t \quad (3.5)$$

The ratio is then multiplied by the advantage of the action, which is the difference between the expected cumulative reward to the value function estimate of the state. This term is known as the unclipped part of the surrogate objective function and can be seen written out in Eq. (3.5). The clipped Surrogate Objective function, has two probability ratios, an unclipped and a clipped which ranges between $[1 - \epsilon, 1 + \epsilon]$ [29]. The clipped objective term ensures that the policy update is within the trust region of the old policy. This is achieved by taking the minimum of the clipped and unclipped objective which is then multiplied by a clipping factor. The clipping factor determines the size of the trust region, and is a hyperparameter that can be defined when tuning the agent.

As seen in Fig. 3.4 the surrogate objective is clipped based on the advantage, if the advantage is positive, it means that the action taken by the agent had a better outcome than expected, meaning it perform well under the current policy and the objective will be clipped at $1 + \epsilon$ to prevent a large action update. If the advantage is negative, it means that the performed action is worse than expected and is more probable under a new policy and the objective will be clipped at $1 - \epsilon$.

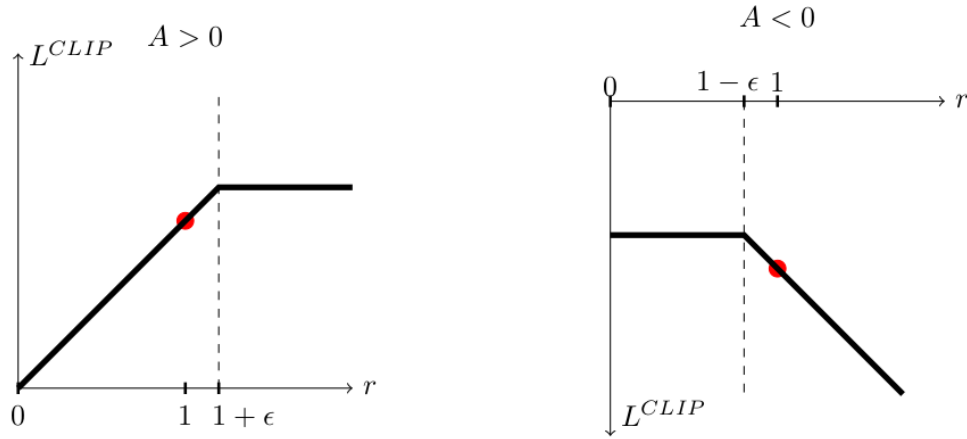


Figure 3.4: The clipped objective function [30]

Hyperparameter:

The following hyperparameters are used to tune an RL agent, where some are common parameters for RL, while others are specific to PPO.

n_envs:

The number of environments running in parallel during training. It is always recommended to use multiple environments to learn faster and to have a robust model [31].

n_steps:

Refers to the number of steps to run for each environment per update [31].

batch_size:

Determines the number of experiences in each iteration of gradient descent. This number should be multiple times smaller than `buffer_size` [31].

gae_lambda:

Is known as the Regularization parameter, which is used to calculate the Generalized Advantage Estimate (GAE). This parameter determines how much the agent will rely on the current value estimate when updating the value estimate. When `lambda` is low the agent relies more on the current value estimate, while if the `lambda` value is set lower it is considered that the agent relies more on the actual rewards received. It is important to find a value that has a good trade-off between the two, to ensure stable training [31].

gamma:

It is the discount factor for the future rewards received by the agent from the environment. This parameter determines how far into the future should the agent care about in regard to future possible rewards. This value is set in accordance to if the rewards are more immediate or further away in the future [31].

normalize_advantage:

A boolean value which is set to true or false depending on whether the advantage should be normalized or not [32].

n_epochs:

Is the number of epochs when optimizing the surrogate loss, one epoch can be defined as one pass through all the training data, which means that all data has had the opportunity to update [31].

ent_coef:

Determines the value function coefficient for the loss calculation. The entropy coefficient is a regularisation parameter, which helps prevent premature convergence, which ultimately hinders the agent in exploring, as one action probability may become dominating for the policy [33].

learning_rate:

The learning rate determines the initial rate of the gradient descent when training starts. This value corresponds to the strength of each gradient descent update step. It is recommended to decrease this value if the increase in rewards is not consistent, or the training is unstable [31].

clip_range:

As previously defined the clip_range determines the size of the trust region, ultimately determining the size of the policy update [31].

The value range for each hyperparameter can be found 3.1.

Hyperparamter	Range
n_envs	-
n_steps	32 - 5000
batch_size	continuous actions: 512 - 5120 discrete action: 32 - 512
gae_lambda	0.9 - 0.95
gamma	0.8 - 0.9997
normalize_adv.	true/false
n_epochs	3 - 30
ent_coef	0 - 0.01
learning_rate	1e-5 - 1e-3
clip_range	0.1, 0.2, 0.3

Table 3.1: The table shows the different hyperparameters that can be tuned, along with their typical value range.

3.3 Simulation Software

The use of simulation to test is preferred for several reasons including safety, cost, reproducibility, and flexibility. This is especially the case when it comes to training and testing RL agents.

There exists many different simulation software with each their capabilities. An analysis is done to determine which software to use. Only software related to solid bodies is considered.

Isaac sim

It is one of the most popular simulation software at the moment. It is proprietary software developed by Nvidia and uses the PhysX engine and has photo-realistic rendering. Its extension Isaac Gym is widely used for RL due to its ability to run the environments in parallel, synthetic data generation, and offload the work to the GPU. It requires an Nvidia RTX graphics card to run it and it supports headless mode [34].

MuJoCo

It is another popular simulation software. It is an open-source software developed by DeepMind and is widely used for model-based control and to test and benchmark RL agents and algorithms. It is accurate, cross-platform, and runs on the CPU. Furthermore, it allows the modeling and programming of the environment using a Python API. MuJoCo uses OpenGL to render its GUI and can run headlessly. RL applications can be developed by using the dm_control software stack which was also released by DeepMind and uses MuJoCo as its physics engine [35].

Webots

It is an open-source robot simulator developed by Cyberbotics Ltd. It is easy to use and cross-platform which allows for fast prototyping. It supports multiple programming languages such as C++, Python, Java, and more. It is gaining more popularity with RL applications with the introduction of OpenDR, which is a DRL platform developed for it. Webots uses a fork of Open Dynamics Engine (ODE) and can run headlessly but requires an X-server to be configured, unlike other solutions. Unlike Isaac Sim and MuJoCo, which only run on GPU and CPU respectively, Webots can run and render using both CPU and GPU [36].

Gazebo

It is an open-source robot simulator developed by Open Robotics. It is widely used in robotics applications with ROS. However, its use for RL applications is limited as it is slow compared to other simulators. It is cross-platform and uses ODE as its default physics engine. It also supports other physics engines such as Bullet and DART. Gazebo (now known as Gazebo Classic) will have its end-of-life in 2025 in favor of Ignition Gazebo which mitigates some of the issues with Gazebo Classic such as strongly coupled software design. Similar to Gazebo Classic, Ignition Gazebo is not widely used with RL applications [37][38].

Simulator	Headless	Hardware Requirements	OS Supported	License
Isaac Sim	Yes	Nvidia RTX GPU	Windows and Linux	Proprietary
MuJoCo	Yes	CPU (x86 or x64)	Windows, Linux, Mac	Open-source
Webots	Yes ¹	CPU (x86 or x64) or GPU	Windows, Linux, Mac	Open-source
Gazebo Classic	Yes	CPU (x86 or x64) or GPU	Linux	Open-source
Ignition Gazebo	Yes	CPU (x86 or x64) or GPU	Windows, Linux, Mac	Open-source

Table 3.2: Comparison between different robotics simulators in terms of the ability to run headlessly, hardware requirements, OS support, and license requirements.

Based on Tab. 3.2, which shows a comparison between the different robotic simulators, MuJoCo has been selected to be used in this project. It was chosen due to its open-source nature and the availability of a software stack that supports running RL agents. Even though Isaac Sim comes with a vast array of features, its hardware and license requirements are the reason why it is not chosen to work with. Both Gazebo Classic and Ignition Gazebo are known to be slower than other simulators and therefore their use with RL is limited and the reason why they have been excluded. Even though MuJoCo and Webots offer similar functionalities, Webots is GUI-bound making running parallel RL agents more complicated and it requires more configuration to run headlessly [39].

¹Requires an x-server to be running locally as it is GUI-bound

Chapter 4

Problem Formulation

As presented in the problem analysis, the conventional lab designs utilized today are outdated making them inadequate to support adaptation of robotics and automation. Thus, the need to re-thinking the current lab setups arises. Today all experimental procedures and lab tasks are done manually by humans and are not directly transferable to a robot, which forms the motivation for self-driving labs. Based on the information gathered from the research on current labs, it was found that the equipment and certain procedures for the preparation of experiments do not allow for the inclusion of robots due to the complexity and intricacy of the task. Therefore it is decided to focus on the planning and execution phases of experiments. This led to the investigation of current research projects, one such project presented is ChemOS. This project attempts to tackle the challenges faced in self-driving labs by designing a software stack that can be used to perform experiments, record data in databases and learn from the results. The robotics module in ChemOS only acts as a bridge between the devices and ChemOS but does not provide any drivers or protocol implementation to ease hardware integration. Other projects, such as the mobile chemist have integrated a mobile manipulator in a lab while providing capabilities similar to ChemOS. However, a significant amount of time was spent on integrating the software automation and communication protocols. The research into skill-based systems gave insight into how to compose skills and create tasks at a higher level. Similarly, current research on the use of RL agents for control contributed to the understanding of how to define the control of shuttles as an RL problem. With this in mind, it was decided to focus on the hardware integration and control of self-driving labs enabling intuitive task planning and execution using behavior trees.

4.1 Final Problem Statement

How can planning and execution of lab-related tasks be done using BTs in the MPS enabling easy programming and system reconfiguration, while achieving control of the shuttles with the help of an RL agent?

4.2 Objectives

Inspired by the presented research works and solutions currently available, the solution presented in this project will focus on the development of a skill-based system using behavior trees for the easy creation of lab tasks. Furthermore, for the control of the shuttles in the MPS system, RL will be used. The objectives are derived from the final problem statement, which is broken into smaller tangible tasks. As the solution proposed consists of multiple components the objectives are divided into 2 parts; one for the skill-based system and another for the RL agent. The objectives are used to

assess the solution in connection with the tests performed in this project.

4.2.1 Behavior Engine

Easy programming of lab task:

The system must provide a user-friendly interface for the visual programming of robot tasks. The system must have a skill library containing lab-oriented skills, which can be combined to create experiments. This can be done by creating a skill-based system using BTs.

Tasks in parallel:

The system must allow the creation of tasks as well as provide the option of performing multiple tasks in parallel. The BT implementation must support the use of parallel nodes.

Feedback:

The system must provide the user with feedback in the form of visualization of the state of the nodes in the behavior trees, during execution. This can be achieved by providing feedback on the user interface in the form of color indicators.

4.2.2 RL Agent

Shuttle control:

The agent must be deployable on multiple shuttles. It must be able to control the shuttles and complete any given task within the reachable space. The PPO algorithm will be used for training the agent and tuned using Weight and Biases. The performance of the agent will be assessed based on time and accuracy.

Collision avoidance:

The agent must complete its task without colliding with other shuttles or going out of bounds. This can be achieved by designing a reward function that penalizes the agent for collision.

Physical setup:

The agent must be transferable to a physical setup and validated on the MPS system. This can be done by using the actions that the agent takes and giving it as an input to the PMC.

Chapter 5

Implementation

This chapter takes the reader through the process of implementing the presented solution. The chapter is divided into the following sections, design, software stack and architecture, device manager, behavior tree, skill library, and RL agent.

5.1 Design

The design section presents an overview of the overall system architecture showing how all the components of the solution are interconnected. This section gives the reader an intuitive understanding of the solution which helps to ease into the following sections that go more in-depth with the technicalities of implementing each component. The second part of the design gives a walk-through of the interface.

5.1.1 System Architecture

Based on the requirements of the system, the following design has been made.

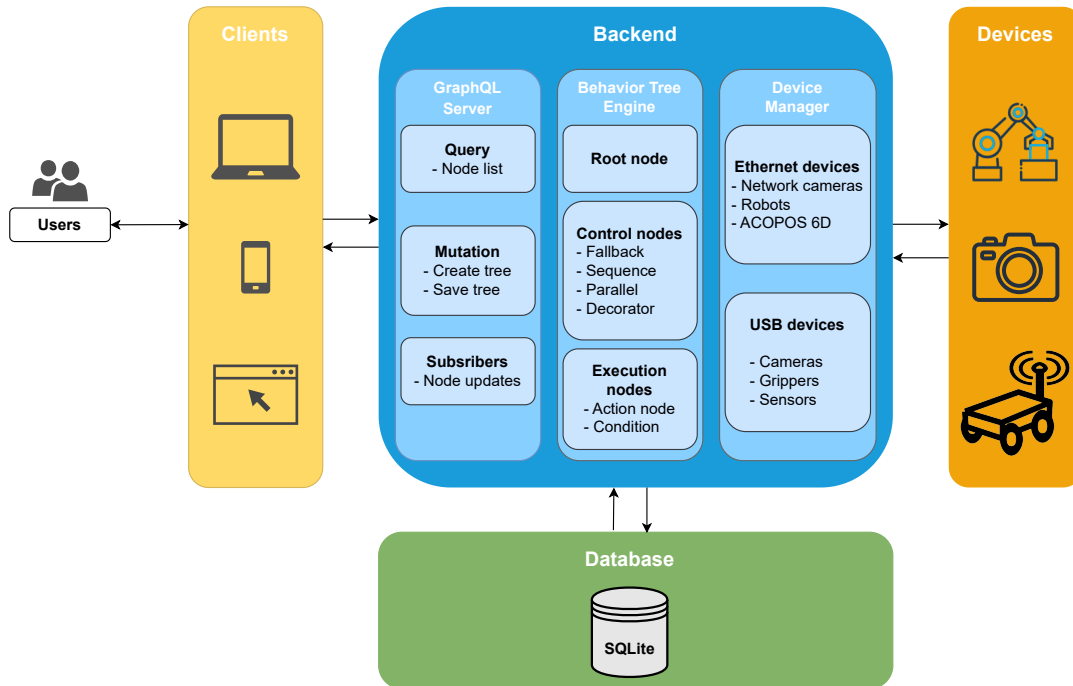


Figure 5.1: System overview showcasing the components of the backend and how data is sent between devices and users.

Fig. 5.1 shows that the system is split into four different parts; clients, backend and devices, and database. The user interacts with the system using the clients to program

the desired device and monitor the system using a behavior tree frontend. The interface runs a GraphQL client which connects to a GraphQL server running on the backend. The GraphQL server is used by the frontend to query, mutate and subscribe to data changes. The backend handles the execution of behavior trees and the interaction with the connected devices. Lastly, a database is used to store the behavior tree and its parameters. The different components are designed based on the single responsibility principle. This allows the system to be modular and the components to be independent in their design and implementation.

5.1.2 User Interface

The user interface for the behavior tree system was implemented based on some of the findings on UI and UX which can be seen in Appendix A.1, combined with knowledge from studying the current solutions and common color trends. Fig. 5.2 shows the overview of the interface. Drawing knowledge from current solutions presented in Chapter 2, the commonality found between them is the simplistic and dark monochromatic interface design. This design seemed to draw attention away from all the unwanted things and focus on the functionality of the system. It was therefore decided to adopt similar design principles for the solution in this project. For the colors, it was decided to go with the color blue, as blue is known for representing calmness, and trust and is many UI designers' go-to color, hence seen in many applications and websites (e.g. LinkedIn, Facebook, Twitter).

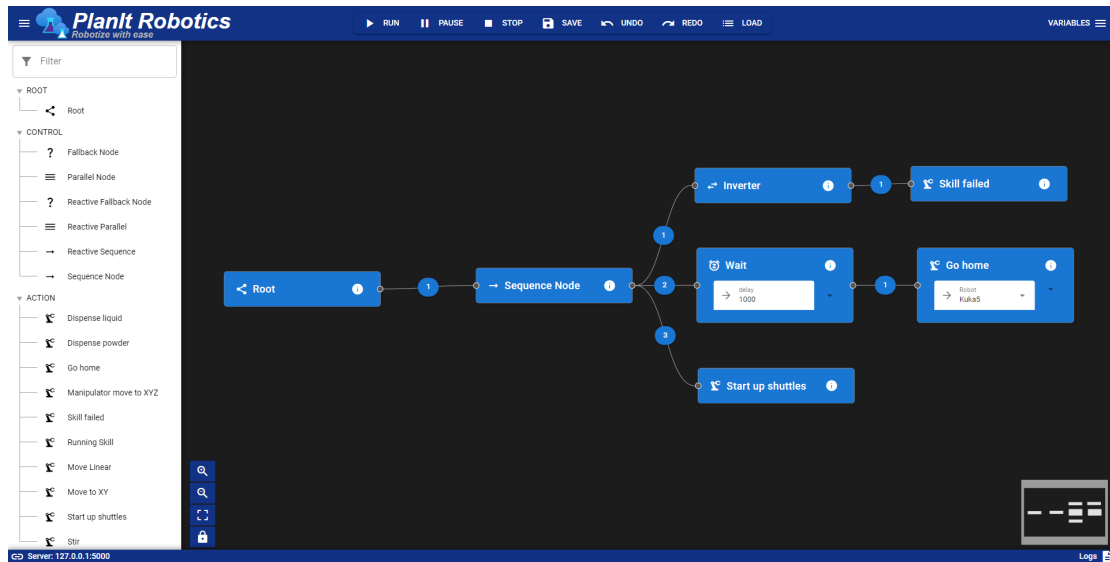


Figure 5.2: The user interface showing the general features, such as the sidebar with nodes, the top menu consisting of buttons to run, pause, stop, save, etc.

Fig. 5.2 shows an overview of the user interface showing all the general features, such as the sidebar with nodes that the user can drag and drop to create BTs, the top menu with all essential buttons (run, stop, etc.) which also contain a load button for loading exciting trees.

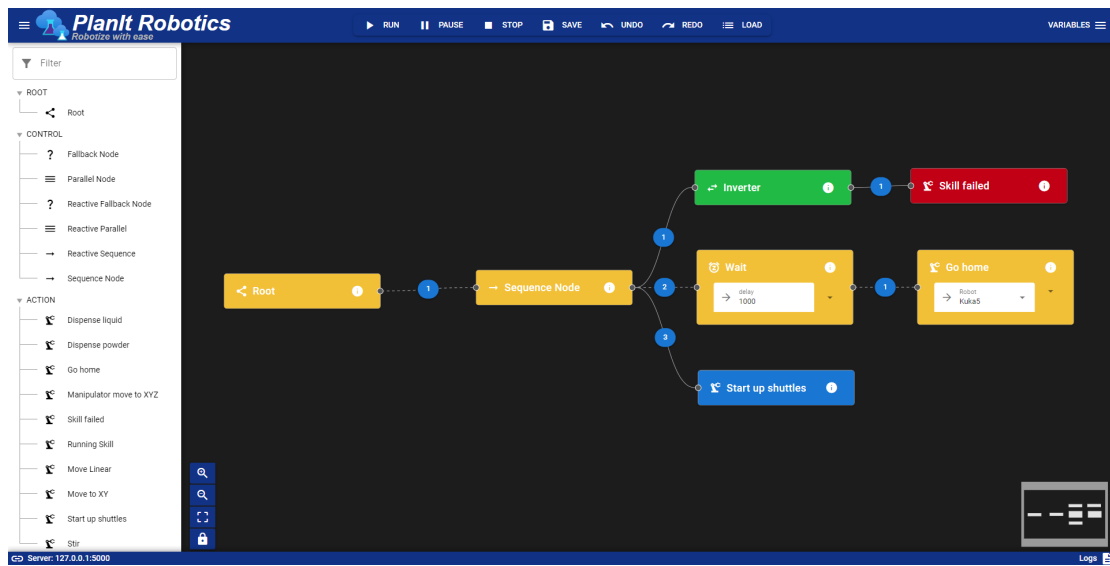


Figure 5.3: Screenshot showing the active colors of the nodes when the BT is running. Yellow for running, red for failure, and green for success.

When running a BT as shown in Fig. 5.3, the colors become active according to the status of the nodes, yellow for running, green for succeeded, and red when the node execution fails. This gives the user an indication of what is occurring during runtime, this also makes it easier to debug in case of errors.

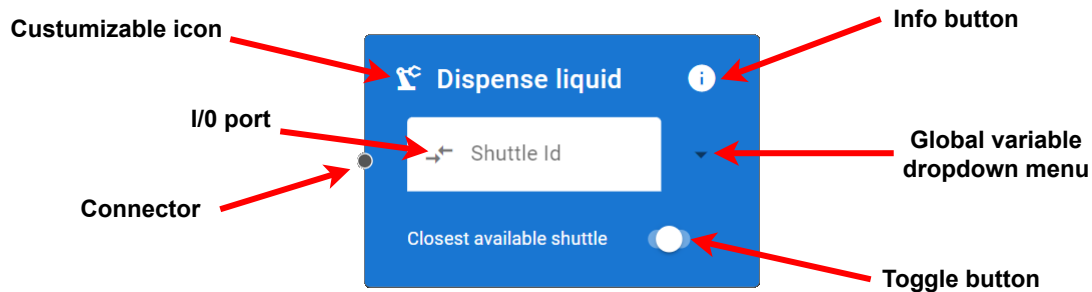


Figure 5.4: The figure shows an example of a BT node that describes all the features implemented, for more details see 5.2 for code snippet.

The node itself 5.4 has many customizable features such as toggle buttons, icons, and I/O port. All of these values are defined in the backend and rendered in the frontend. See 5.2 for a code example.

5.2 Software Stack and Architecture

The main part that the user interacts with is the interface of the system which is referred to as a behavior engine. This section describes the software stack and architecture used in the backend and frontend.

5.2.1 Backend

The backend is built using ASP.NET which is a cross-platform framework for building server applications using .NET and C#. It provides access to services in a simple API to add functionalities to the server, such as database management. It also allows managing custom services in an inversion of control containers [40]. This makes it possible to use the dependency injection design pattern as the system's capabilities differ based on the configured devices.

The responsibility of the backend is to provide a node list of the behavior tree that the system is capable of running based on the configured devices and services. All nodes inherit from an `INode` interface making it possible to determine which nodes should be sent to the frontend for rendering. This is done using a computer science principle known as reflection which allows the program to examine objects and determine types at runtime.

The backend provides different services that can be used to perform a variety of tasks. Some services include protocols to communicate with PLCs such as ADS (Automation Device Specification) and OPC UA (Open Platform Communications Unified Architecture) protocol. Furthermore, the backend is used to run behavior trees and has an implementation of different nodes, and allows the user to create their own nodes. Lastly, the backend is used to create different databases based on a code-first approach using Entity Framework [41].

The behavior tree implementation is designed in such a way that it is possible to create a tree and run it from the code using a Fluent API. This means that the user can run a behavior tree without the use of the frontend. Alternatively, the user can create a behavior tree using the frontend which sends the tree structure to the GraphQL server running on the backend.

A sequence diagram, seen in Fig. 5.5, shows the communication between the server and client, along with the devices connected to the system.

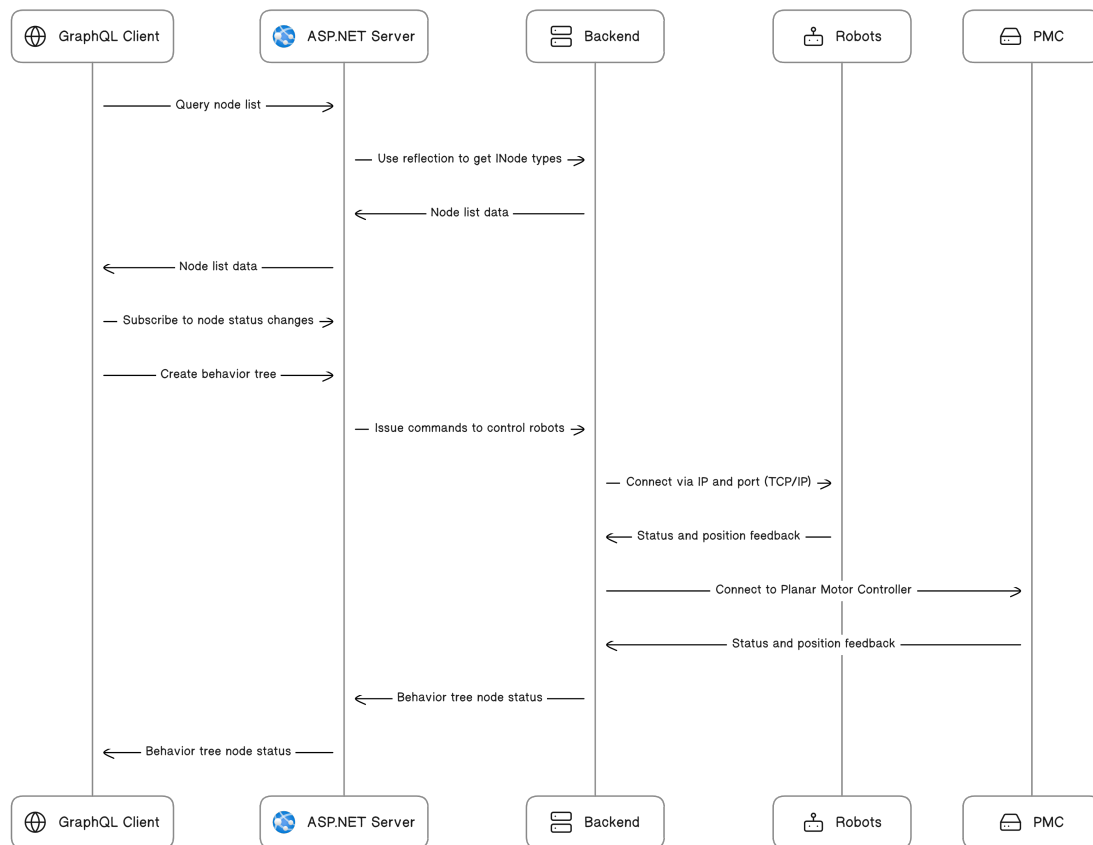


Figure 5.5: Sequence diagram showing the communication between the GraphQL clients with the ASP.NET server. The diagram also shows the connection to the different devices along with the data provided by them.

GraphQL is chosen to design the API as it is more flexible than other APIs such as REST. Unlike REST APIs which request predefined data models, GraphQL makes it possible to request the exact data needed making it more efficient. Furthermore, GraphQL supports subscribing to data changes such that it will send updates to the clients [42].

5.2.2 Frontend

It is desired to have a frontend that supports multiple platforms, giving the user the option to work on existing hardware. This can be achieved in different ways, one of which is to write different native clients for each platform (Windows, Linux, Mac, Android, iOS). The other way is to use web technologies to make a cross-platform client which can be used with any different platform. Of the two options, the latter was chosen as the client is written only once, saving time and making the code more consistent with all the clients [43].

For this JavaScript, HTML, CSS, and Vue.js have been used. Vue.js is an open-source framework used to develop web applications and user interfaces [44]. Vue.js uses a model-view-viewmodel architecture which allows the creation of reusable components. Furthermore, its two-way data binding feature, which automatically synchronizes the

data and the interface whenever either of them changes, is one of the reasons why Vue.js has been chosen to work with rather than other frontend frameworks such as React. To enable the use of multiple desktop platforms (Windows, Linux, and MAC), Electron.js is used. Electron.js uses the Chromium web browser engine to render the user interface and is used widely in the community to build cross-platform applications [45].

To support mobile devices such as Android and iOS phones and tablets, Capacitor is used. Capacitor acts as a bridge between the web application and the Webview of the native operating system of the mobile device. Both Electron and Capacitor provide APIs to allow the user to access the devices' capabilities such as cameras, file systems, notifications, and more [46]. The frontend is the main part that the user interacts with. It provides the user the ability to create, run, load, save, and debug behavior trees in addition to giving feedback on its status.

5.3 Device Manager

The device manager handles the connections to the devices and robots by providing clients and protocols to send and retrieve data. To work with the ACOPOS 6D shuttles, a client was made supporting OPC UA. An OPC UA server must be configured in the PLC with desired authentication method. Moreover, the variables of interest must be configured in the server to expose them to the clients. The client, which runs on the backend, is used to send commands to the shuttles. It has methods to read, write and subscribe to the exposed variables by specifying their location (path and namespace) in the server [47].

As this setup requires the configuration of the OPC UA server and designing data structures to store and manipulate the shuttles, the Planar Motor API is used instead. Planar Motor provides C# and Python APIs to control the shuttles by giving access to the Planar Motor Controllers (PMCs). The system also supports the use of the XPlanar system which was developed by Beckhoff Automation. A library to communicate with the PLC that controls the XPlanar shuttles were created. Details about this library are included in Appendix A.5.

To control the robots, RoboDK has been used. The lab setup has been modeled in RoboDK as shown in Fig. 5.6. The different robots along with the tabletop have been added to the environment to ensure that the robot's trajectory is collision-free. Moreover, different reference frames have been set in the environment such as a base frame for each robot as well as a reference frame similar to the shuttle position reference. This simplifies the transformation of the shuttle position from the shuttle reference to the robot reference frame. RoboDK can find the inverse kinematics solutions for a given pose in a given reference frame. This way, the robots can be controlled to move in the Cartesian space with the same coordinates as the shuttles.

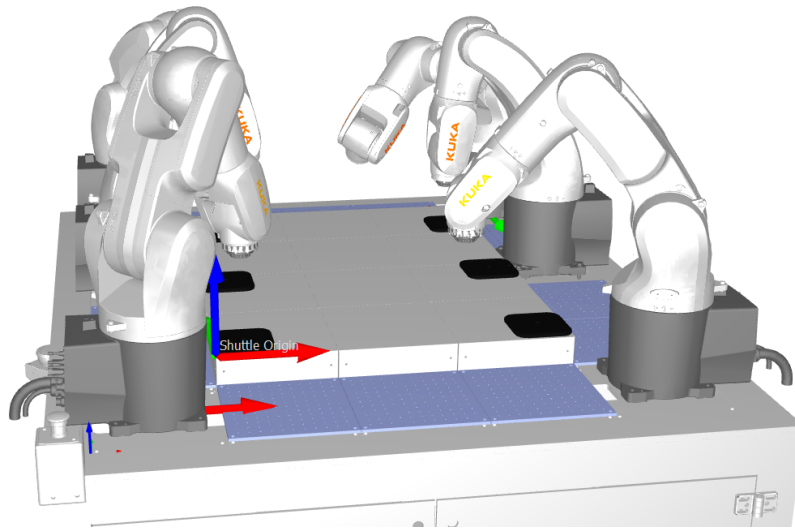


Figure 5.6: Lab setup in RoboDK showing the KUKA robots, shuttles along with the shuttle reference frame which is placed in the bottom left corner of the ACOPOS platform.

RoboDK requires a TCP/IP server to be running on the robot controller. Using the RoboDK API, it is possible to connect to each robot using its IP address and port [10].

To support the vision system using Ethernet cameras, the GigE Vision interface is supported. This interface is standardized and allows interfacing with various brands of Ethernet cameras. It has multiple features such as the automatic discovery of cameras on the network, device configurator, and requesting image streams [48]. In addition to supporting the GigE Vision interface, the system has support for Intel RealSense cameras. Intel provides SDKs for accessing their cameras and enabling different streams. In this project, the Intel RealSense D435 is used, which has two streams for RGB images as well as depth images. Similar to GigE Vision, the RealSense cameras can be discovered when plugged into the system using the SDK.

5.4 Behavior Tree

Unlike other behavior tree implementations, which give the user the responsibility to create blackboards (dictionaries to store and retrieve variables) and manage them, this implementation makes it seamless. Here, each node has its local blackboard called *link* which is used when values are saved as an input. Furthermore, a global link is created and shared among all the nodes. The global link is used to save the values of the output and the values of the inout port. When an inout port is used, the ID of the node is concatenated with the key and saved in the global link allowing other nodes to access it and modify it. Similar to MoveIt Studio, the conditions nodes and action nodes are separate. This makes it possible to check the condition once in a sequence where multiple nodes have that specific condition. This also means that the condition node can be used as a pre or post-condition depending on where it is positioned in the tree.

```

1 public interface INode
2 {
3     Guid Id { get; set; }
4     string Name { get; }
5     string Description { get; }
6     string Icon { get; }
7     NodeType Type { get; }
8     NodeCategory Category { get; }
9     NodeState State { get; }
10    int MaxChildren { get; }
11    Action<Guid, NodeState> OnNodeStateChanged { get; }
12    INode Parent { get; set; }
13    NodeLink LocalLink { get; }
14    List<LinkPort> Ports { get; }
15    List<INode> Children { get; }
16    Task TickAsync(TreeLink globalLink);
17 }

```

Listing 5.1: The INode interface which all BT nodes inherit from.

Listing 5.1 shows the INode interface which all BT nodes inherit from. It can be seen that additional properties are defined such as name, description, and icon. These properties are used when rendering the frontend which is sent during runtime. Each node has a state which can be success, failure, running, or idle. It can also be seen that each node has an event that gets triggered whenever the node state is changed. This event can be subscribed to allowing sending feedback on the current node execution, which is done to send data to the GraphQL clients. Furthermore, each node must specify a list of ports, which are used to render the node in the frontend. Based on the type of variable in the port, different options are shown to the user.

```

1 public override string Name => "Dispense liquid";
2 public override string Description => "Station for dispensing liquid";
3 //public override string Icon => "precision_manufacturing";
4
5 public override List<LinkPort> Ports => new()
6 {
7     new LinkInOutPort<int>("Shuttle Id", "Shuttle id", "id"),
8     new LinkInputPort<bool>("Closest available shuttle", "closest shuttle", "
    ↪ isClosest"),
9 };

```

Listing 5.2: Code snippet of the dispense liquid node showing the name, description, icon, and ports.

Listing 5.2 shows an example of a dispensing node. It shows that one port is created with the type int and another with the type bool. In the frontend, they will be rendered as an input box and a toggle button, respectively, as shown in Fig. 5.4. Type checks will be performed on the ports ensuring that the other types cannot be inputted. Enum types will be rendered as a dropdown menu showing the enum values.

Similar to other implementations of BT, different categories are defined. The categories are root, control, decorator, and execution. Here, different classes implement

the `INode` interface defining the common functionality in the specified categories. This includes defining how many children the node in the specific category can have. It also includes defining events to allow debugging the trees among other things. For the control nodes, sequence, fallback, parallel, reactive sequence, reactive fallback, and reactive parallel nodes have been implemented. The reactive nodes are the nodes that are ticked to ensure reactivity. They differ from the normal nodes by the continuous ticking of their children when the tick is propagated to them. An example of the tick function for the sequence node can be seen in Algo. 1. Other implementations for different nodes are shown in Appendix A.4.

Algorithm 1: Implementation of the Tick function for a sequence node

Data: `globalLink`

```

1 async Task TickAsync(globalLink):
2   State ← NodeState.Running;
3   await Children[CurrentChildIndex].TickAsync(globalLink);
4   switch Children[CurrentChildIndex].State do
5     case NodeState.Success do
6       | CurrentChildIndex++; break;
7     end
8     case NodeState.Running do
9       | State ← NodeState.Running; return;
10    end
11    case NodeState.Failure do
12      | State ← NodeState.Failure; return;
13    end
14  end
15  if CurrentChildIndex ≥ Children.Count then
16    | CurrentChildIndex ← 0;
17    | State ← NodeState.Success;
18  end
  
```

Similar to BehaviorTree.CPP¹ used in MoveIt Studio, this implementation is asynchronous by default. This is beneficial when working with parallel nodes which allow executing different actions and conditions simultaneously. Even though the node is parallel, it is not truly parallel. A true parallel would require a CPU core for each child the node has, making it use all system resources quickly.

¹<https://www.behaviortree.dev>

5.5 Skill Library

The behavior tree itself helps to orchestrate different actions. However, a well-thought action and condition library is needed to help the user accomplish their desired tasks. Based on the objectives the system must accomplish, different actions and conditions nodes were created. This section goes in-depth with the different skills and the reason for implementing them. Skills other than the ones mentioned in this section are also available. These skills expose lower-level commands such as moving to x and y for a specific shuttle as well as moving to x, y, and z for manipulators.

5.5.1 Decorators

Some functionalities are desired in many different applications. For example, inverting an output, repeating a task, retrying when failing as well as waiting for a specified amount of time. These are known as decorators that allow the user to create more efficient and resilient behavior trees. For example, it is possible to check for a condition when a specific value must be met to succeed. However, if it is desired to check that a condition is not true, an inverter can be used. The inverter node inverts a failure status to success. The retry node is used to retry an action or condition for a number of times that can be specified. When the child node returns success, then the retry node also returns success. If the number of retries is exceeded, the node returns failure. Unlike the retry node, the repeat node ticks a child node multiple times. A flowchart of the repeater node can be seen in Fig. 5.7. The wait node is designed to be asynchronous, allowing the user to wait without blocking the thread. This is useful when combined with a parallel node.

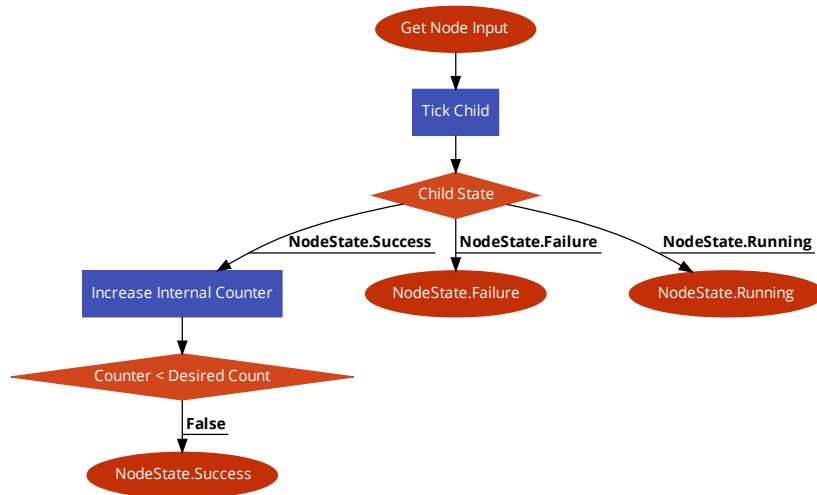


Figure 5.7: Flowchart of the tick function implementation of the repeater node.

5.5.2 Start-up Shuttles

To be able to communicate with the Planar Motor Controller, a connection has to be made to it. This skill runs a procedure starting by connecting to the controller, then

gaining mastership of the system to allow sending commands and controlling the shuttles. This is done as only one master can use the system at a time. Afterward, the shuttles have to be activated if they are disabled. If any shuttle is moving, it will be stopped. When all shuttles are active and not in motion, then a command is issued to start levitating. These steps are done using an action node that has no input. If any of the steps fail, this node fails. A flowchart of the startup node can be seen in Fig. 5.8.

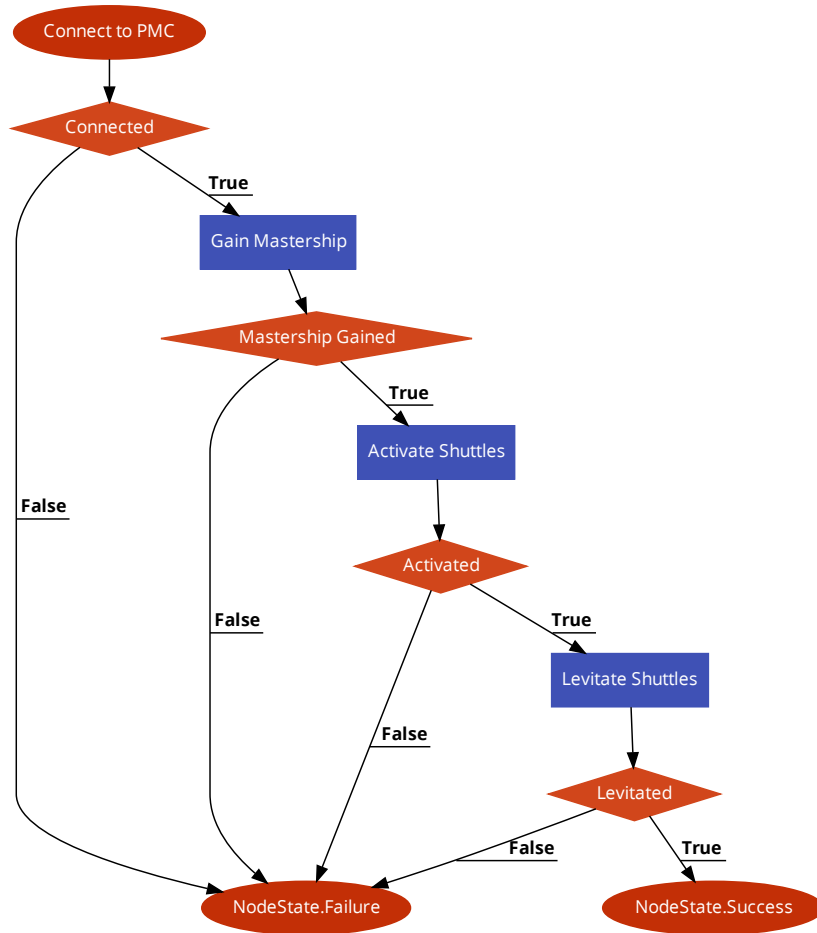


Figure 5.8: Flowchart of the tick function implementation of the startup node.

5.5.3 Homing

Before moving the robots to different positions, it may be preferable to home the robots. This node has a dropdown to select a robot among the available ones. Even though there are two types of robots, the joint configuration for their home position is the same. This is why it was chosen to home the robots using joint space by providing the angles for each joint. The angles are sent to RoboDK which creates a trajectory to move the robots to the desired joint configuration. A flowchart of the homing node can be seen in Fig. 5.9.

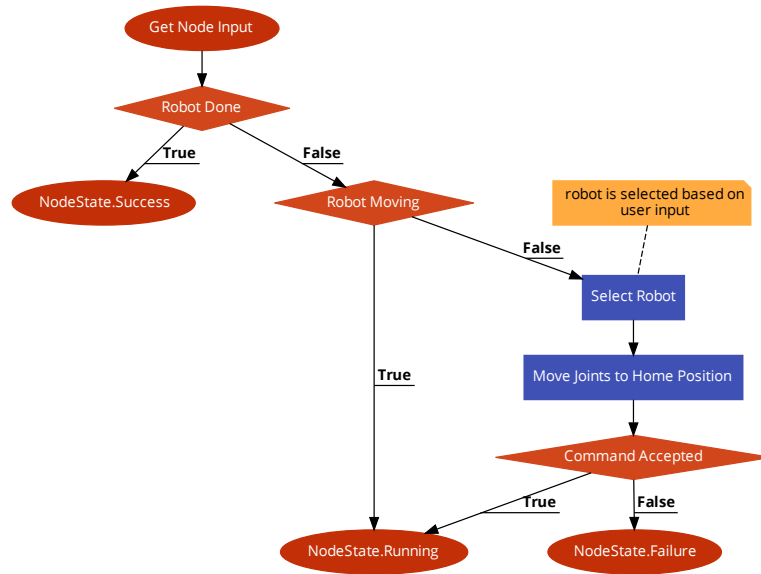


Figure 5.9: Flowchart of the tick function implementation of the homing node.

5.5.4 Stirring

This skill is used to rotate a shuttle, selected by the user, with a desired frequency for a specified amount of time. The designer chooses a rotational frequency, indicating the number of rotations per second, as well as a designated duration for the rotation. The skill uses a functionality in the Planar Motor API which is used to rotate a shuttle with a specific rotational velocity and duration. A flowchart of the stirring node can be seen in Fig. 5.10.

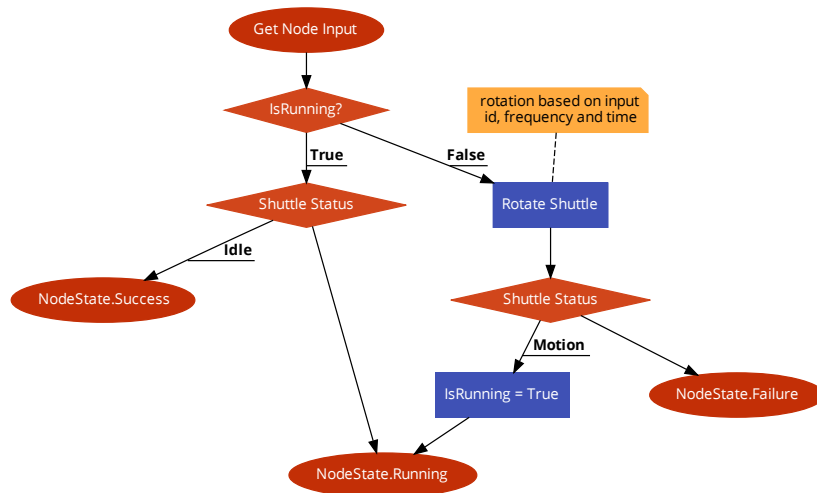


Figure 5.10: Flowchart of the tick function implementation of the stirring node.

5.5.5 Dispensing

As the test setup is not an actual lab, the dispensing is not performed. However, to imitate that a robot is dispensing, the robot moves to a position above the shuttle. The user can choose a shuttle by ID, or select the nearest shuttle automatically. The chosen shuttle then moves to the designated station using the Planar Motor API. When the shuttle has reached its goal, the robot gets the current position of the shuttle and moves in Cartesian space to its location with a specific height. The orientation of the gripper is chosen to be perpendicular to the shuttle. To transform from the shuttle reference frame to the robot's reference a homogeneous transformation matrix is used, the derivation of the matrix can be found in Appendix A.6. When the robot reaches the shuttle, a waiting function is used to imitate the time it takes to dispense. A flowchart of the dispensing node can be seen in Fig. 5.11. Two different dispense functions were created, one for liquid and one for powder. The two skills only differ from each other by the waiting time and the station location. If the robot or the shuttle fails to get to the desired position, the action fails, otherwise the action succeeds.

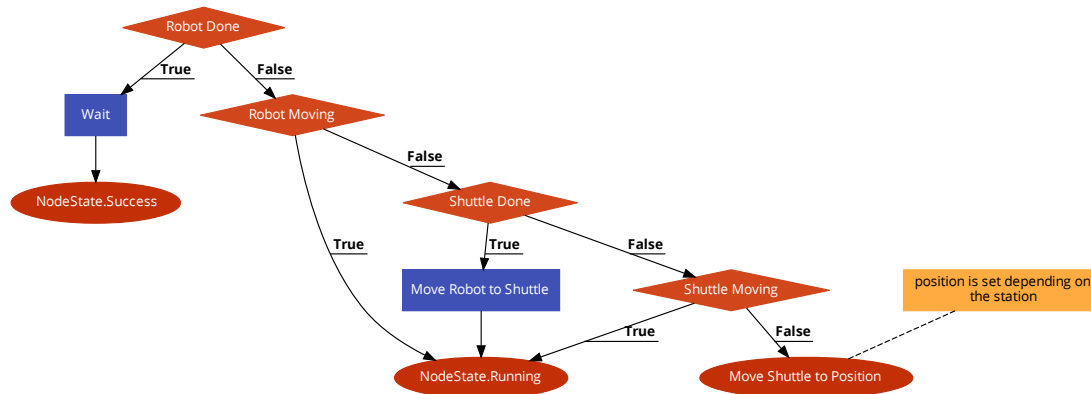


Figure 5.11: Flowchart of the tick function implementation of the dispense skill.

5.5.6 Color Detection

At the visit to the chemistry lab, the extent of task automation within experiments was discussed. It was found that during certain experiments some chemical compounds change color when mixed and this color change may have an indication of the progress of the experiments. Mixtures that turn yellow or brown are usually a sign of something wrong or that the experiment is unsuccessful. It was therefore decided to implement a vision skill that can be added in the BT when planning a task that can detect these color alterations. It was decided to use a simple color detection algorithm with 3 colors to choose from, either yellow, red, or blue. This is done by using OpenCV (Open-source Computer Vision Library) and defining the color range based on the HSV color space seen in Fig. 5.12. HSV consists of Hue which determines the color, and Saturation tells how strong the color is, which is commonly defined as how much gray is mixed in the color. Value is the brightness level of the color.

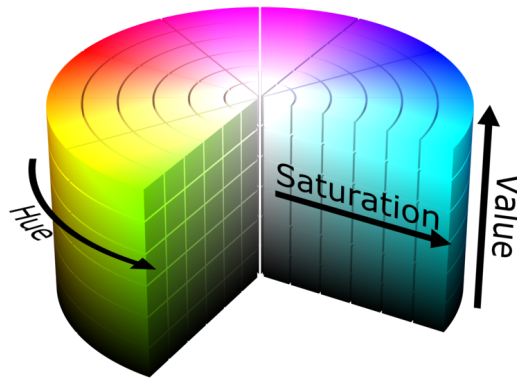


Figure 5.12: Figure illustration of the HSV color spectrum defined in degrees.[49], ranging from 0 to 360.

HSV is defined in degrees ranging from 0 to 360, however, OpenCV color ranges from 0 to 180 to fit into the uchar datatype which only encodes numbers from 0 to 255. So any color defined is divided by two, to get the color range in OpenCV [50]. A flowchart of the color detection node can be seen in Fig. 5.13.

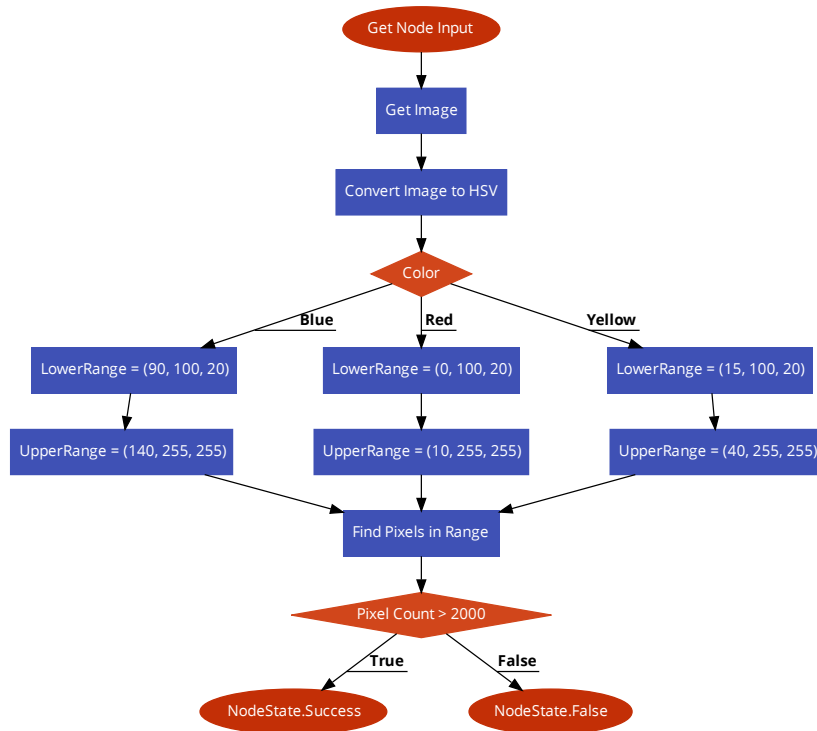


Figure 5.13: Flowchart of the tick function implementation of the color detection condition.

The color detection algorithm approach simply converts the RGB image from a real-sense camera into HSV color space to detect the specified colors. The vision node allows the user to choose a color from a dropdown menu, which detects and filters away

any other color in the image. In Fig. 5.14 the first image shows the actual RGB image followed by the color detection of the chosen colors, blue, red, and yellow, respectively. The vision skill has proven feasible for real-time detection of colors and can therefore be used to monitor experimental procedures.

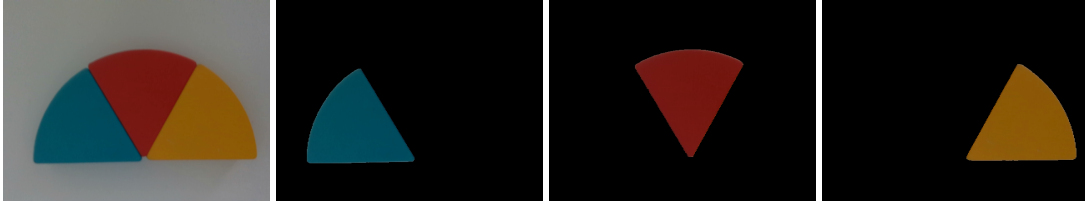


Figure 5.14: Figure showing the RGB image from the real-sense camera, along with the color detection images for each color detected.

5.6 Reinforcement Learning Agent

For the training of the RL agent, PPO with Stable-baselines3 was used. Stable-baselines3 is an open-source library that contains different RL algorithms and tools that can be used for training [51]. The library is built upon PyTorch which is widely known within the machine learning community. Stable-baselines3 comes with many features which ease the training of reinforcement learning agents. It complies with the OpenAI Gym environment structure making it compatible with many environments. Furthermore, it has integration with TensorBoard, which is a visualization tool that helps in understanding the model and its performance by looking at the logged metrics. In addition to that, it has integration with Weights and Biases, which is a tool similar to TensorBoard but with other benefits such as parallel training for the purpose of automatic hyperparameter tuning.

The RL agent interacts with the environment created, through the rewards and observations it receives. To understand their relationship, the observation space and action space have been described in this section. In addition to these two components, the reward function for the agent is determined. It was decided to define the problem as one RL agent controlling all shuttles, commonly referred to as a multi-armed bandit problem, compared to the alternative of defining it as a multi-agent system. An agent for both discrete actions and continuous action will be implemented to analyze how they learn during training.

5.6.1 Discrete Case

This section describes the thoughts and processes that went into the implementation of the discrete case, divided into the essential components of the RL agent, namely the observation space, action space along with the reward system.

Observation space

The observation is based on the environment being described as a grid system, due to the actions being discrete. In Fig. 5.15 below, the grid world is visualized. The figure

displays a hypothetical environment based on the ACOPOS system where the goal is represented as a red dot, and the shuttles are shown as green squares. It is considered that at each timestep the shuttles move 1 grid space. The grey spaces marked with an x are the padding added to the environment, so the agent always has the same number of actions.

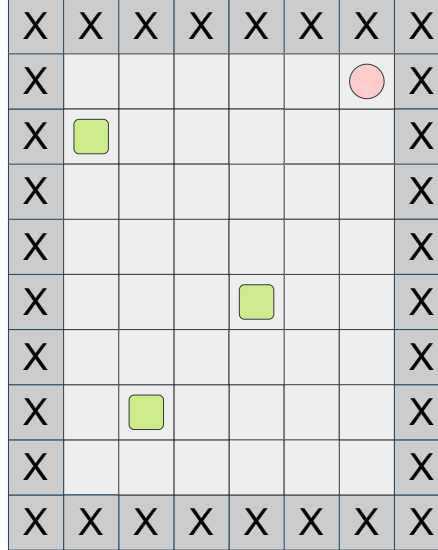


Figure 5.15: Figure illustrating the environment for the discrete actions which is based on a grid world. This environment is the basis for the observation space given to the agent.

The observation space consists of 6 values, the first 4 being the availability in the different directions (up, down, left, and right), 0 indicates that the grid space in the corresponding direction is available. If the value is 1, it means that the space in that direction is occupied due to another shuttle being there or the shuttles being next to the boundary. The last two values of the observation space are a directional vector from the current position of the shuttle to the goal, one value in the x direction and one in y.

Action space

Initially, the intention was to have one agent control all shuttles during training, but this led to poor results and did not give representative data of the shuttles' individual performance as all rewards were pooled together as a sum of rewards. This gave a false representation of the performance which made it hard to analyze and locate the problem in the shuttles' behaviors. Due to this, it was decided to train one single agent and use the other shuttles as static obstacles. This model could then, in theory, be deployed on each shuttle making them able to reach their assigned goal, while avoiding other shuttles placed in the environment.

In the first iteration of the action space, it was determined to have 9 actions available for the agent, as it was meant to be able to move in all directions both straight and

diagonally, and have the option to stay in its current position. Even though in the discrete actions, the agent can avoid obstacles when moving diagonally, in real life it would collide with the obstacle. Another issue is that the distance moved diagonally is larger than the distance moved along x or y. This makes it more complex to transfer the learned agent to the real setup.

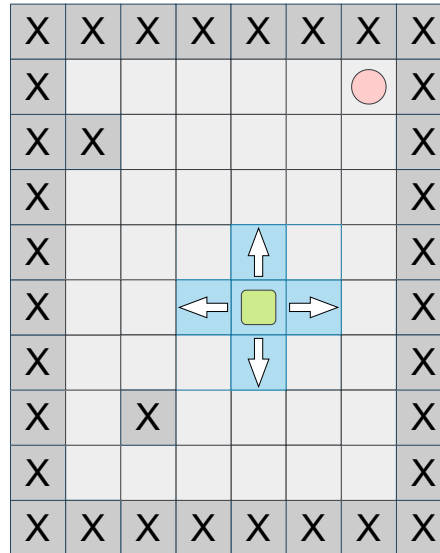


Figure 5.16: Illustration of the action space for the discrete actions. These are the actions that the agent can choose from at each timestep.

In the following and final approach, the action space was reduced to 5 possible actions, as illustrated in Fig. 5.16, restricting the shuttle's actions, so that they will be unable to move diagonally. This helped alleviate the problems described above. At each timestep, the agent will send one action to the shuttle to perform. This value would be a number between 0-4 as seen in Fig. 5.17, each corresponds to an action (up, down, left, right, and stay).

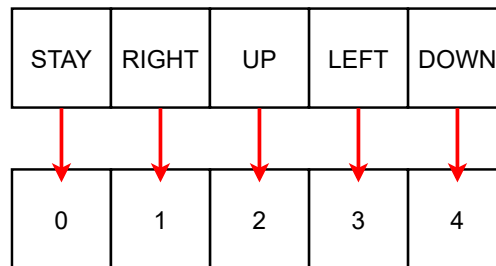


Figure 5.17: Actions which can be taken by the agent; up, down, left, right, and stay. Each action corresponds to a number value which is sent to the shuttles as a command.

Reward system

The first implementation of the rewards given to the agent was based on the principles of a dense rewards system. When implementing a dense reward system, it was observed that during training the agent would try and move to the goal position given regardless of obstacles, meaning that it would ignore the collision penalty and move straight towards the goal. This caused the shuttle to repeatedly collide with the obstacles it encountered on its path, which resulted in the length of the episodes being very short, not giving the agent time to learn or explore. It was speculated that the behavior was caused by the high reward assigned when reaching the goal, so it was decided to reduce the completion reward (reaching the goal) and to increase the penalty for a collision. With the revised rewards the agent learned to avoid the edges and other obstacles. But as the reward for reaching the goal was not too high and the penalty was now higher than said reward the agent prioritized collision avoidance. It is observed that the agent did not seem to move towards the goal but just avoided obstacles at all cost. In some cases the episode never ended as the agent did not collide nor did it complete its task, and the training had to be terminated manually. It was then decided to add a small positive reward for moving closer to the goal to motivate the agent to move and complete the task. This had a very negative impact on the training, as the agent became greedy and decided to not end the episode and was collecting all the immediate rewards it received for closing in on the goal, without going to the goal to end the episode.

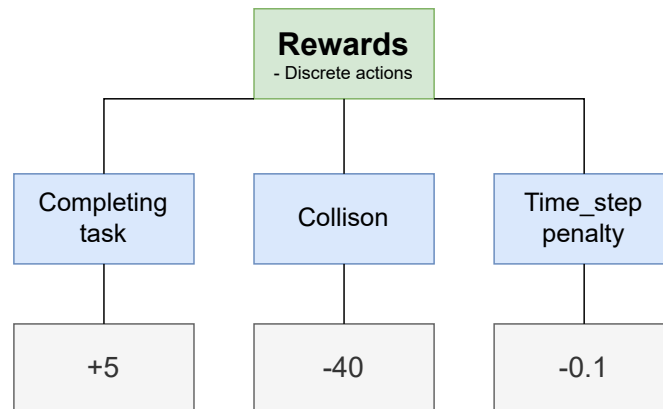


Figure 5.18: Final rewards for the agent based on discrete actions. The rewards are based on a sparse reward system.

After a few more iterations, it was finally decided to change to a simple sparse reward system, as it was considered that having too many types of rewards and penalties was a distraction for the agent, and hindered learning. The rewards defined for the sparse reward system can be seen in Fig. 5.18. The training results achieved with the final rewards are documented in Chapter 6.

When running the real shuttles it is desired to not have any rigid movements or unstable trajectories, which can occur when using a model which was trained based on discrete actions. Therefore, continuous actions are preferred as it relates better to the physical setup.

5.6.2 Continuous Case

This section delves into the implementation of the agent with continuous action space. Unlike the discrete case, the continuous case environment is built with MuJoCo using Python bindings. The positions of the shuttles and the goals were randomized to give a more robust model that can adapt to changes. Furthermore, it was decided to set the number of shuttles to four to reduce the complexity of the training.

As it is desired to deploy the trained model on the actual system, the capabilities of the Planar Motor system had to be investigated. The Planar Motor API connects to the PMC and allows the user to access the IDs, positions, and statuses of the shuttles. Additionally, the API allows sending position commands in the form of x and y coordinates along with a max velocity, max acceleration, and an end velocity. Therefore, the observation space and action space will be influenced by these factors.

Observation space

The observation space is defined to let the agent learn to go to the desired goal while avoiding collision with other shuttles and boundaries. A directional vector from the current shuttle position to the goal is given in the observation space. This vector is calculated as follows:

$$\vec{v} = \begin{bmatrix} x2 - x1 \\ y2 - y1 \end{bmatrix} \quad (5.1)$$

Additionally, the relative position of the other shuttles is also given in the observation space enabling the agent to learn when it gets close to another shuttle. This makes a total of six values, two for each shuttle other than the one being controlled.

The observation space also includes the distances to the boundaries of the environment. As discussed previously, the ACOPOS 6D system setup is moving along motor segments, adding a constraint to the agent, hence this observation. The distances are represented as follows:

$$boundary_{max_x} = max_x - position_x \quad (5.2)$$

$$boundary_{min_x} = position_x - min_x \quad (5.3)$$

$$boundary_{max_y} = max_y - position_y \quad (5.4)$$

$$boundary_{min_y} = position_y - min_y \quad (5.5)$$

Lastly, the difference between the current action and the previous action is given as an observation. This allows it to observe the rate of change of the velocities to help prevent oscillation.

Action space

The action space is represented as a vector of size 2 defined in the range of -2 and 2. This range has been chosen so the agent can determine the direction of where shuttle's movement. Furthermore, as the maximum speed of the ACOPOS 6D system is 2 m/s, the bounded range of 2 is selected.

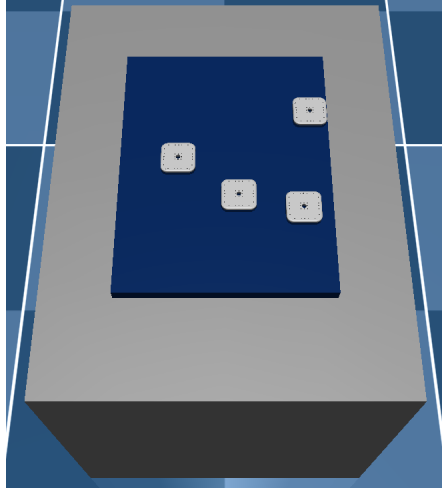


Figure 5.19: MuJoCo environment showing the simulation of the ACOPOS system used for training the reinforcement learning agent.

In MuJoCo, when an environment is created, a controller can be set which has access to all the objects in the environment. The action taken by the agent is given to the controller and is used to change the velocity of the shuttle.

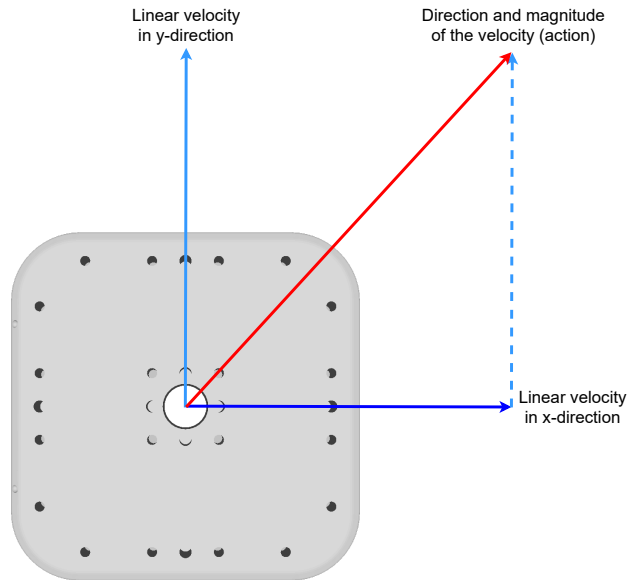


Figure 5.20: Shuttle model showing the action, representing linear velocity in x and y. The vector in red shows the sum of the two action vectors which determines the direction and the speed.

The shuttle model showing the action taken by the agent can be seen in Fig. 5.20. The figure shows the two actions, in blue and light blue, representing the linear velocity in x and y, respectively. The sum of the two vectors is shown in red determining the direction and the speed of the shuttle.

Reward system

Several rewards have been defined in the environment to train the agent to reach its desired goal, avoid collision and getting out of bounds. Both sparse and dense rewards have been defined in the environment. Different weights were defined for the reward functions to adjust their importance. A reward is given to the agent for reaching its desired goal. This reward is defined as:

$$r_{gr} = \omega_{gr} \quad (5.6)$$

To motivate the agent to move towards its desired goal, a dense reward based on the distance to the goal was defined, seen in Eq. (5.7).

$$r_{gc} = \exp \left(-\frac{1}{2\sigma^2} (\|\mathbf{P}_{shuttle} - \mathbf{P}_\star\| + d_g) \right) \quad (5.7)$$

Both σ and d_g are constant values used to adjust how steep the slope is for getting to the goal. $\mathbf{P}_{shuttle}$ and \mathbf{P}_\star are 2-dimensional vectors representing the positions of the shuttle and the goal, respectively. $\|\cdot\|$ denotes the norm of the relative distance to the goal calculated as $\mathbf{P}_{shuttle} - \mathbf{P}_\star$.

Another dense reward was defined to encourage the shuttle to end the episode as quickly as possible by penalizing each timestep. This reward is defined as:

$$r_q = -\omega_q \quad (5.8)$$

In this case, it was decided to set ω_s to 1, which makes 0 the maximum dense reward the agent can receive since the goal attraction reward has a maximum of 1.

Additionally, colliding with other shuttles and going out of bounds is penalized. This reward is defined as follows:

$$r_{col} = -\omega_s \quad (5.9)$$

The agent is also penalized for getting close to an obstacle if the distance to the obstacle is closer than a defined range.

$$r_{ob} = \begin{cases} \exp \left(-\frac{1}{2\sigma^2} \cdot (d_{obstacle} + d_g) \right) & \text{if } d_{obstacle} < 0.3 \\ 0 & \text{otherwise} \end{cases} \quad (5.10)$$

Where $d_{obstacle}$ is the distance to the obstacle (shuttle). This is calculated for each shuttle and the reward is summed.

Since the actions are defined as velocities, it is important to make sure that the agent gives stable output so no oscillation occurs. Therefore, sudden changes in the actions taken are penalized. The penalty is defined as follows:

$$r_{os} = \begin{cases} -\omega_{os} & \text{if } \|a_{\text{current}} - a_{\text{previous}}\| < 0.1 \\ 0 & \text{otherwise} \end{cases} \quad (5.11)$$

Where a_{current} is the current action and a_{previous} is the previous action taken by the agent. Furthermore, to ensure that the speed does not exceed 2 m/s, the agent is penalized if the action taken exceeds that. This is done in a similar way to the oscillation penalty as shown below:

$$r_{ms} = \begin{cases} -\omega_{ms} & \text{if } \|a_{\text{current}} - a_{\text{previous}}\| > 2 \\ 0 & \text{otherwise} \end{cases} \quad (5.12)$$

All of these rewards and penalties makes up for the reward function, seen in Eq. (5.13).

$$r_{\text{total}} = r_{\text{gr}} + r_{\text{gc}} + r_{\text{q}} + r_{\text{col}} + r_{\text{ob}} + r_{\text{os}} + r_{\text{ms}} \quad (5.13)$$

5.6.3 Tuning

The tuning of the different parameters has been done with Weights and Biases. Weights and Biases offer a Sweep functionality, allowing the user to sweep the search space for the parameters defined in a configuration file. In this case, the configuration chosen is listed below:

Parameter	Values
learning_rate	[0.0005, 0.001]
gamma	[0.99, 0.9]
n_steps	[64, 128]
ent_coef	[0.01, 0.001]
clip_range	[0.1, 0.2, 0.3]
gae_lambda	[0.9, 0.95]
n_epochs	[1, 5, 10]
batch_size	[16, 32, 64]

Table 5.1: Sweep Configuration

Here, it was decided to try the random method where the weights will be sampled randomly based on a uniform distribution. This is selected in favor of the other methods (grid and Bayes) as the grid method will iterate over all combinations of the defined configuration. The Bayes method was excluded as it requires a metric to be defined to optimize it. The sweep configuration makes it possible to select only one metric such as

reward or episode length. In this case, it is desired to maximize the reward and shorten the episode length, therefore, it is decided to run the random method.

5.6.4 Deployment

To deploy the agent model on the real system, it had to be exported first to another format that can be read in C#. Therefore, the agent has been exported to ONNX format and the ONNX runtime was used to run the agent by loading the exported model.

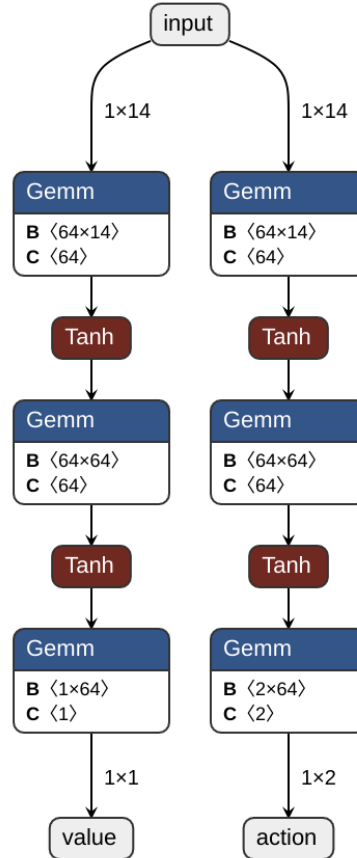


Figure 5.21: A view of the exported neural network showing the inputs, outputs, and their sizes along with the operations performed.

As seen in Fig. 5.21, the model expects an observation of size 14 in the same order as it was defined in the agent. This observation is then saved as a tensor and inputted into the network. The critic network then outputs the expected cumulative reward (Q-value) while the actor outputs the actions. The actions, representing velocities, were integrated to get the x and y coordinates that the shuttle must move to. The norm of the actions was then used to set the end velocity for the shuttles. Both the maximum velocity and maximum acceleration were defined similarly to the trained agent. The maximum velocity is 2 m/s and the maximum acceleration is 20 m/s^2 .

A BT action node has been implemented which uses this model to command the shuttle

to move to its desired goal. The node accepts a shuttle ID as an input along with a desired goal in the form of x and y coordinates.

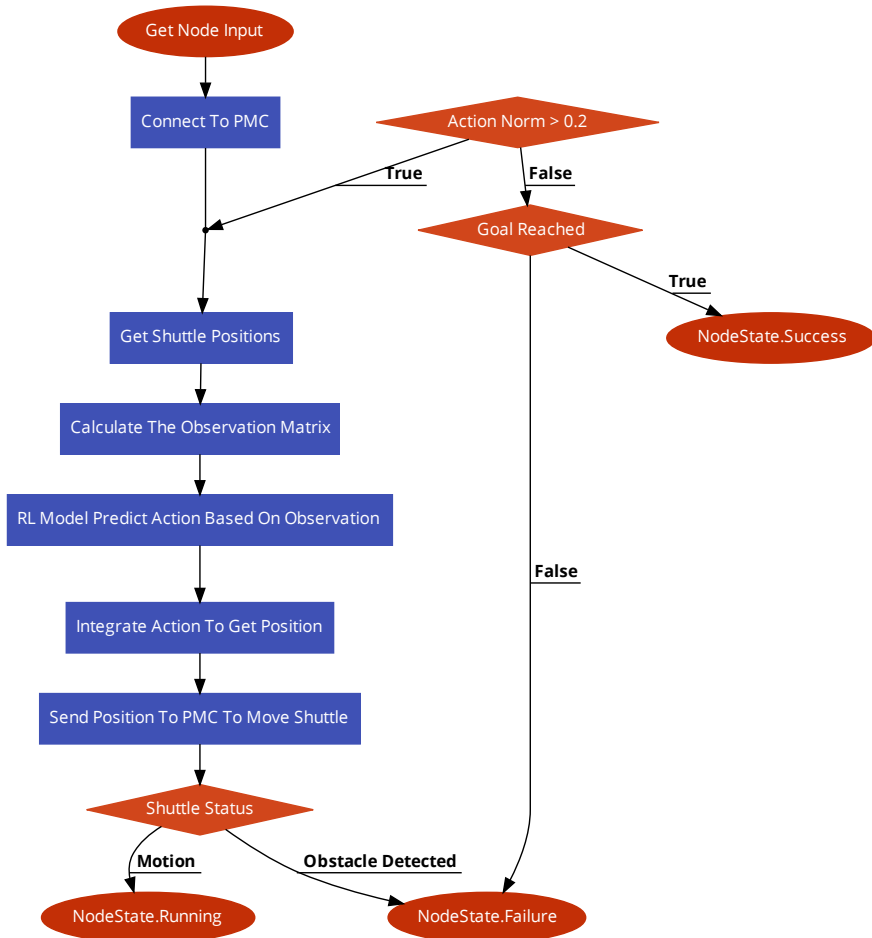


Figure 5.22: A flowchart showing the implementation of the RL agent node used to command the shuttle.

Fig. 5.22 shows how the RL agent node is implemented in the BT. It shows that the velocities given by the agent get routed through the PMC which is used for positional control. The PMC gives access to the position of all the shuttles. These positions will be used to create the observation matrix which will be inputted to the RL agent to predict the action the shuttle should take. This will be done until either the action is too small, indicating the agent is at the goal, or a collision occurred indicating failure.

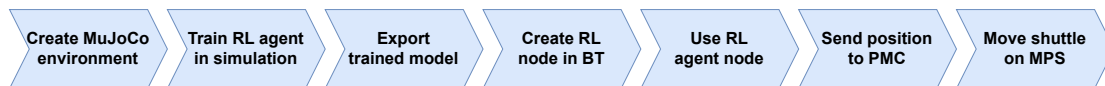


Figure 5.23: Pipeline of going from simulation to deployable RL model using BT.

With all the implementation now described, the full solution pipeline going from simulation to deployment, as seen in Fig. 5.23, is presented.

Test and Results

This chapter will cover the results of the RL agent training as well as the test cases for both the behavior engine and the deployment of the RL agent on the MPS. The training results presented are divided into two part, discrete case and continuous case. This is followed by the tests performed, which are divided into 2 main parts, behavior engine, where the BT system is tested, and deployment of RL agent.

6.1 Discrete Case

The discrete case was the first step in the implementation which was treated as part of the learning process on how to train the agent and construct rewards. The initial testing with no obstacles is presented, followed by training and testing in an environment with obstacles. The results from the training with no obstacles and with obstacles are both shown through the mean episodic reward, which is averaged over each 100 episodes. These results are presented in graphs further below.

6.1.1 Initial Testing

This section describes the process and outcome of the initial testing phase with having one single shuttle and training it with different goals. For each episode in the training, a new random goal and start position is given to the shuttle, as seen in the example in Fig. 6.1.

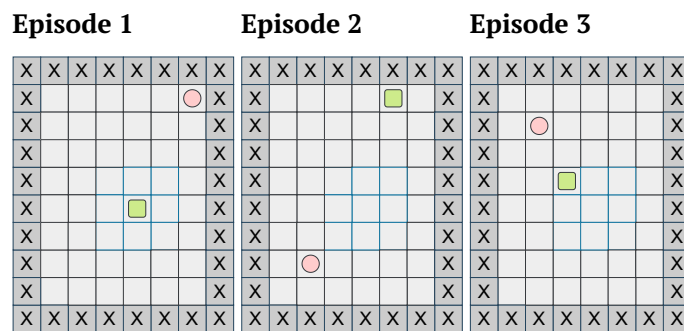


Figure 6.1: Figures illustrating examples for each training episode, where the shuttle and the target goal are assigned at random locations, based on x and y coordinates.

Training and tuning

Fig. 6.2 shows the test performance of the different runs during the initial training, with one shuttle and no obstacles, based on the mean reward. The graph shows 3 different plots of training, one with the initial hyperparameters values and 2 tuned ones. The graph has a smoothing factor of 75, which makes it easier to compare them, visually.

As it can be seen in the graph 6.2, even though that the all runs depicted are different from each other, the commonality the agents shared was the fast learning, converging at around the 40k timestep mark and was considered a sufficient model past said point. The difference between them was the stability of the training. These runs aimed to find parameters that gave fast learning while still being stable during training, avoiding large dips in the rewards received.

Training and Tuning - Episodic Reward

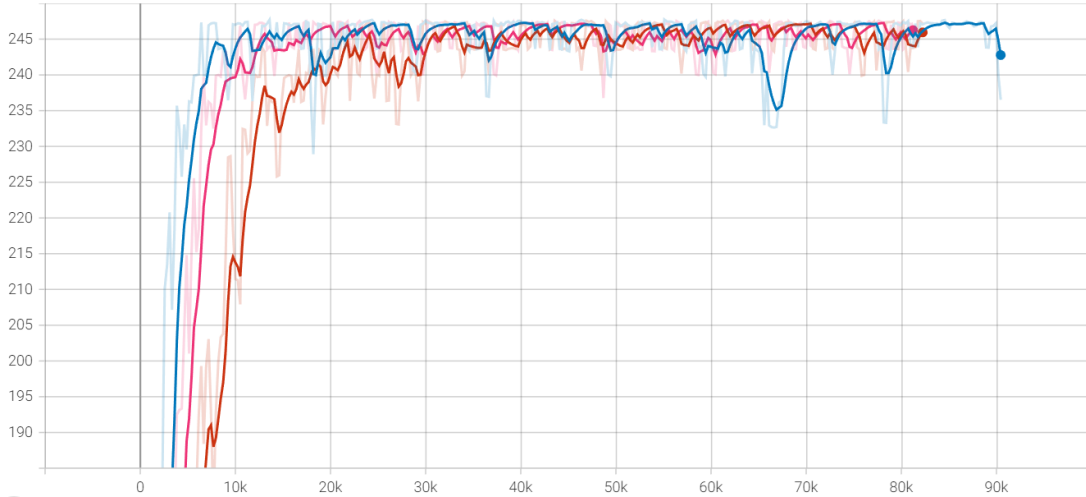


Figure 6.2: Training sessions based on the episodic mean reward, one with initial values for the hyperparameters (blue), and 2 tuned ones (red and pink).

HYPERPARAMETERS								
Runs	n_steps	batch_size	gae_lambda	gamma	n_epochs	ent_coef	learning_rate	clip_range
●	32	64	0.92	0.98	20	0.001	0.001	lin_0.2
●	32	32	0.92	0.99	20	0.001	0.0001	lin_0.2
●	32	64	0.92	0.99	20	0.001	0.0005	lin_0.2

Figure 6.3: Hyperparameters values chosen during manual tuning of the agent.

In Fig. 6.3 the table shows the corresponding hyperparameter values used for each training. The values in the first training (blue) had a high learning rate which made it learn fast and converged sooner than the other two but performed unstably causing large dips in the rewards received. The values decided upon tuning were grounded in the desire to have a more stable and reliable model. As can be seen in the second training (red), the learning_rate, which determines the rate of change at each timestep, was reduced from 0.001 to 0.0001 to have a more conventional learning curve. Gamma was also increased to ideally let the agent weigh the future rewards more, which is more beneficial for it since the rewards are based on a sparse system. This yielded some desired effects on the training and the agent learned more steadily as can be seen in the rewards achieved, but was slower at arriving at the maximum mean reward and to converge. To keep the stability of the second run while still aiming to learn faster, the

learning_rate was slightly increased and the gamma and the other parameters remained the same, the performance of these hyperparameters can be seen depicted in the last run (pink). These were the values that were decided to proceed with for the next phase of training with obstacles in the environment, as it had a preferable trade-off between stability and fast learning.

The results from the training (pink) are shown statistically in Fig. 6.4. The bar chart shows the number of times that the shuttle went out of bounds resulting in the episode terminating, against the number of times the shuttle succeed to reach its goal. It can be seen that the shuttle goes out of bounds several times in the beginning, but improves rapidly in avoiding the edges. The model converges to an optimal model after the halfway checkpoint in the training, which conforms with the graph in 6.2, which shows that the model converges around 40k.

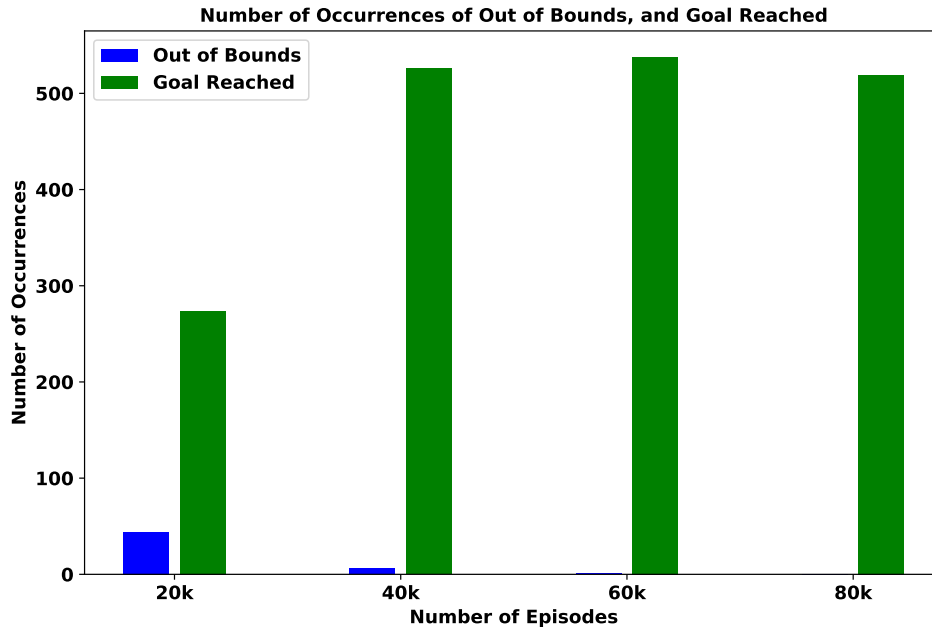


Figure 6.4: Training results with discrete actions and no obstacles. The bar chart shows the number of times the agent went out of bounds and the number of times that the agent reached the goal.

6.1.2 Obstacles

In the following training phase, random shuttles were put into the environment to act as obstacles. These were static obstacles and had no properties of their own, but were only for training the shuttle in obstacle avoidance, which is one of the objectives set for the agent in Chapter 4. At each timestep, a random number of obstacles between 2-10 were generated and assigned a random position in the environment. This was done to ensure a robust model, that could avoid obstacles. The rewards used for the training

with obstacles are the ones stated in Section 5.6.1.

The graph in Fig. 6.5 shows the performance of each model with obstacles. As labeled, 0-NO is the training model from the initial testing that had no obstacles in the environment. 1-B acts as a baseline as it is the first run with obstacles and just one environment, 2-ME is the same training consisting of the same values for the hyperparameters but with multiple environments running in parallel. The last training 3-ME-T is the same as 2-ME but with tuned hyperparameters. By reading the graph, the model that is considered to perform the best is 3-ME-T with obstacles, given the reduced training time to converge and almost reach the maximum reward. The model reached a mean reward of approximately 3.5 out of a maximum of 5. The results from this training model can be seen below.

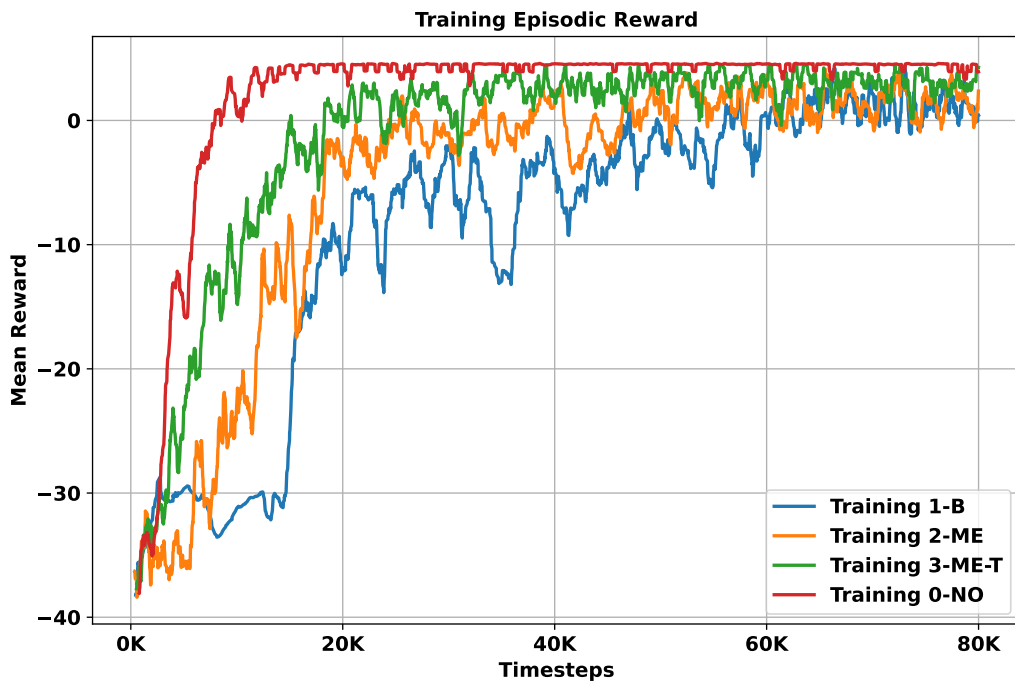


Figure 6.5: The different training are labeled as 0-NO) No Obstacles, 1-B) Baseline, 2-ME) Multi-Environment, and 3-ME-T) Multi-Environment Tuned.

As previously shown in the bar chart of the model trained with no obstacles, the graph below, in Fig. 6.6 shows the statistics for model 3-ME-T.

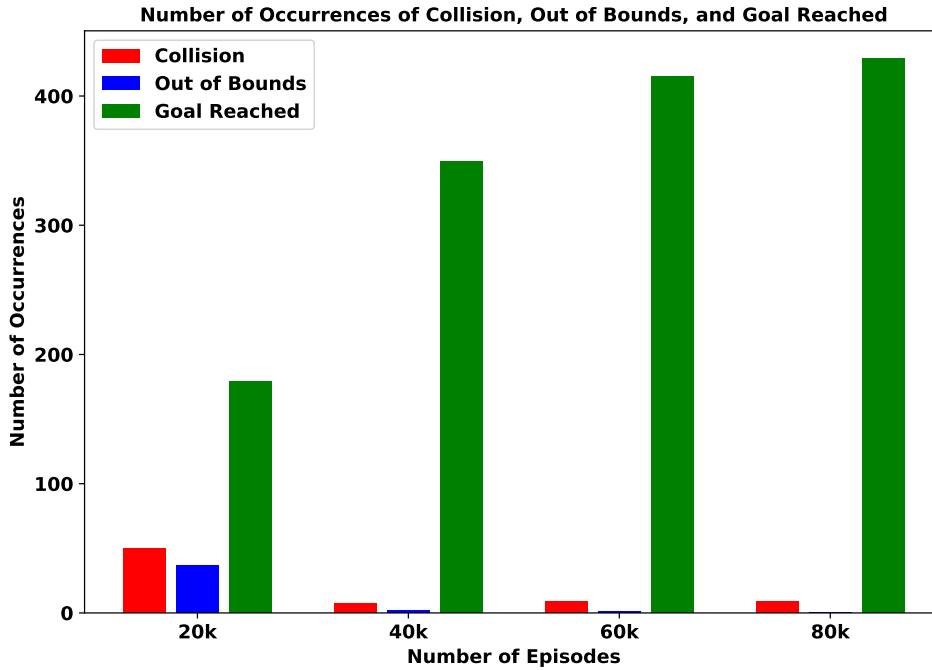


Figure 6.6: Training results based on the number of times the agent collided, was out of bounds, and reached the goal. The figure shows statistics at every 20k checkpoint.

The chart shows the number of times the shuttle collided with other shuttles (red), how many times it went out of bounds (blue), and lastly how many times it succeeded to reach the goal (green). As seen through the training the agent learns very early on in the training that it should not go out of bounds. Through the training the agent improves on avoiding the other shuttles but does not achieve the maximum reward. This is due to the agent still receiving some collision penalties, at the 80k checkpoint, showing that the agent struggles to avoid the other shuttles in some cases.

Fig. 6.7 shows the trained model navigating to its goal at points (5, 4) without colliding with the obstacles. The environment, represented by a matrix, has been plotted with Matplotlib. The dark blue cells show the boundaries as well as the other shuttles. The agent is represented by a yellow cell moving inside the matrix. The agent can only move on the cyan-colored cells. The trained models show that an agent can be successfully trained to navigate and avoid other shuttles. This shows the feasibility of having an RL agent to control shuttles to desired positions, which can be used in lab-related tasks.

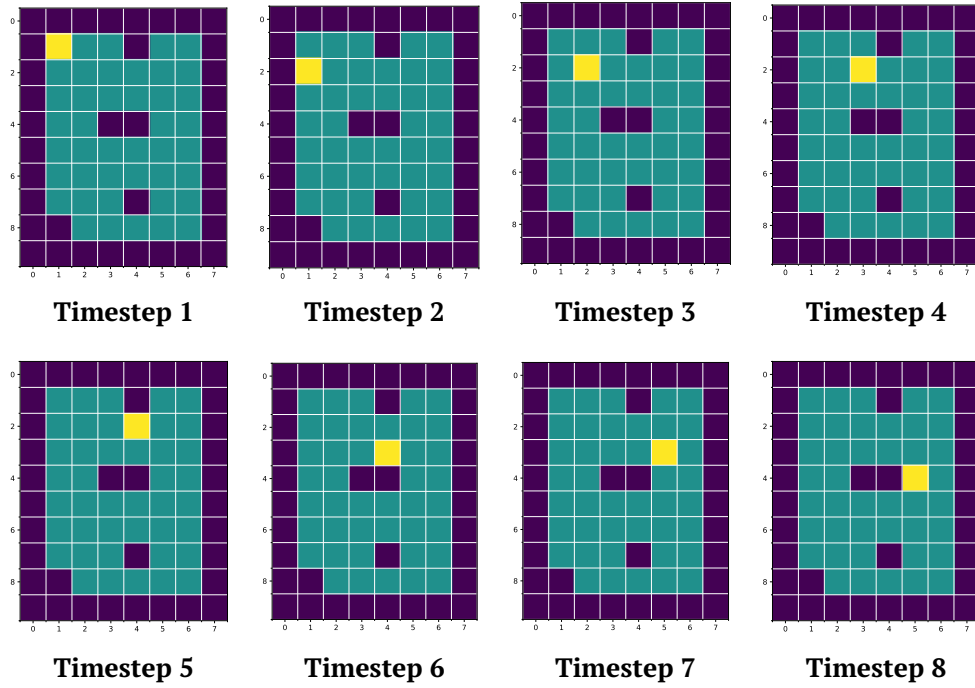


Figure 6.7: An example of the trained agent moving to its goal position at points (5, 4) as seen in timestep 8. The agent successfully reaches its goal without colliding with obstacles. The agent is represented by the yellow cell while the obstacles are represented by the dark blue cells.

Using discrete actions on the physical system would not provide the desired level of control, which is why it was decided to improve on the current concept and move forward with continuous actions. The following sections show the results obtained from the training of agents with continuous actions.

6.2 Continuous Case

The continuous case of the reinforcement learning agent, which uses the environment simulated in MuJoCo to train the agent, is more suitable for deploying on the real setup. The results of the tuning using Weights and Biases are shown in this section. Based on those results a model was selected for testing the agent on the MPS setup. Unlike the discrete case, where the number of shuttles were generated at random, here only the initial position of the shuttles and the goal position were randomized. The training, in this case, consist of 4 shuttles, one being controlled by the agent, while the others act as obstacles.

6.2.1 Training and Tuning

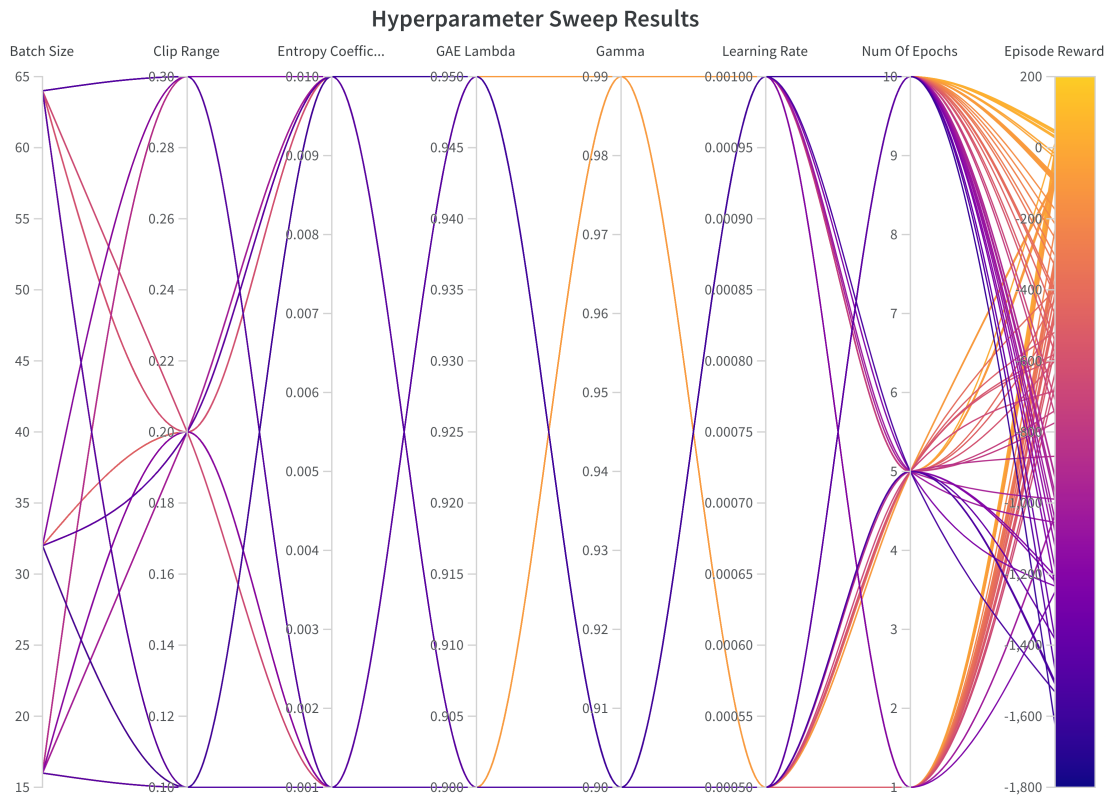


Figure 6.8: Results of the hyperparameter sweep obtained from Weights and Biases showing different hyperparameters along with the mean reward.

Fig. 6.8 shows the results of the hyperparameter sweep. One of the things to notice is that when the value of Gamma is high, the mean reward is also high. The other parameters did not affect the mean reward as much. This corresponds well from the findings in the discrete case.

This can also be seen when looking at Fig. 6.9 showing the impact of gamma on the mean reward. It also shows that the episode mean length is inversely proportional to the reward mean. This correlation is logical as the design of the reward penalizes the agent for taking more time to reach the desired goal.

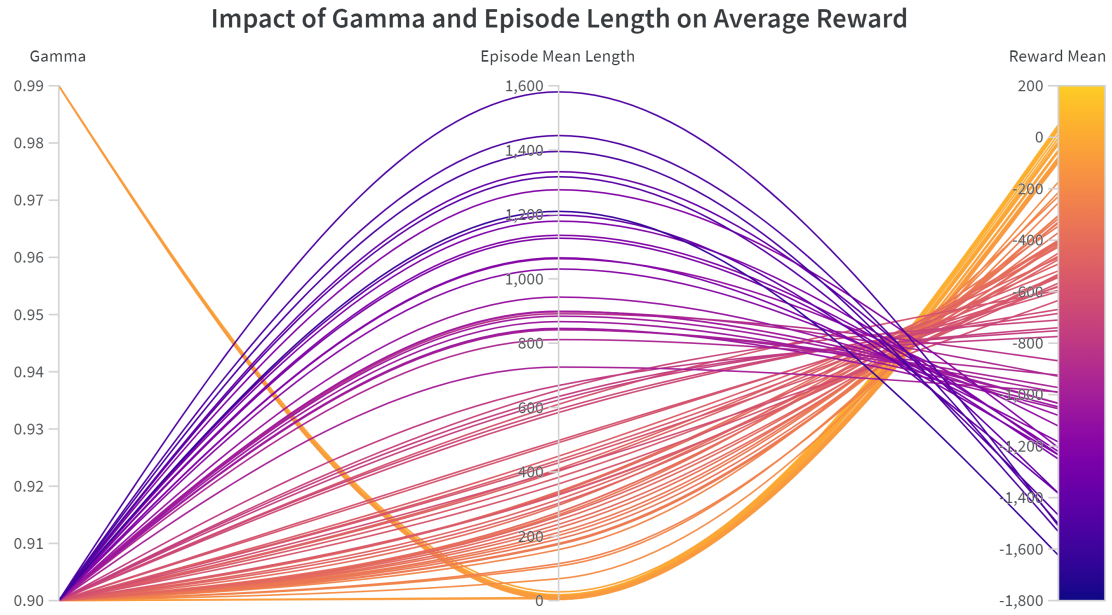


Figure 6.9: Impact of the value gamma on the episode mean length and mean reward.

The value of gamma represents the discount factor, which determines the importance of the future reward compared to the immediate reward. The results indicate that the agent requires longer-term planning to achieve its goal. This aligns with the objective of making the agent.

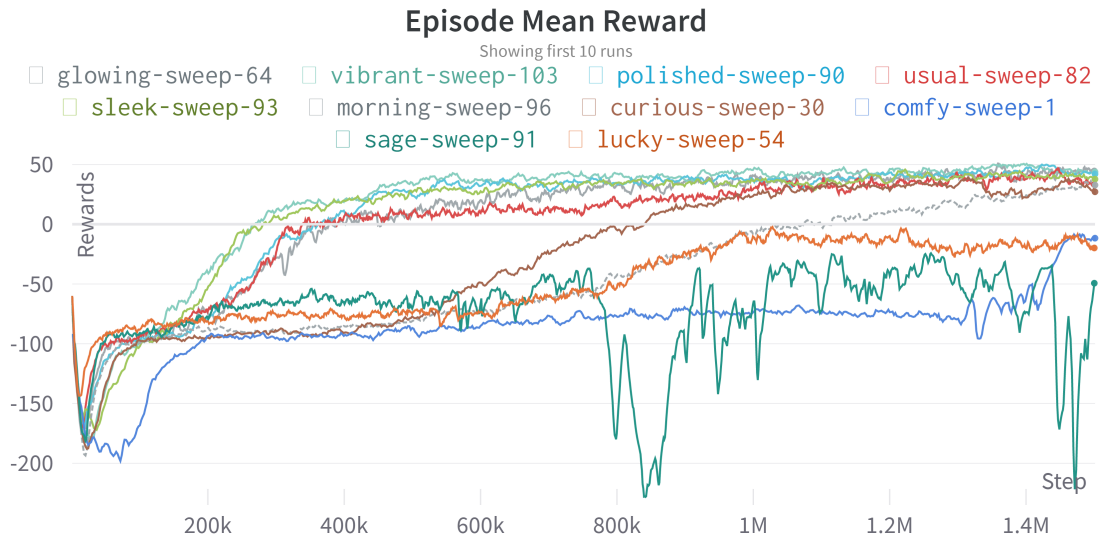


Figure 6.10: Episode mean reward at different timesteps of the best 10 runs.

To select a model to test and deploy on the real system, the results of the runs were compared. Fig. 6.10 shows the rewards of the 10 best models at different timesteps. It can be seen that sweep 64, 103, 90, 82, and 93 converge to a high reward much faster than the other models. Therefore, these models will be tested against each other to see

which one is best to avoid collision and going out of bounds.

6.2.2 Model Performance

The results from the sweeps showed only the reward mean as a metric to determine which model to use. Fig. 6.11 shows the results from the 4 best performing models, found during tuning, based on the number of collisions with other shuttles, out of bounds, and goal reached. The result of sweep 90 has been omitted as it was similar to sweep 64. The results show small differences between the models where sweep 93, outperforms the other models when compared to the number of collisions with other shuttles, however, it lacks when compared to going out of bounds. A screen recording of the performance of sweep 64 in simulation can be seen in this [link](#).

Number of Occurrences of Collision, Out of Bounds, and Goal Reached

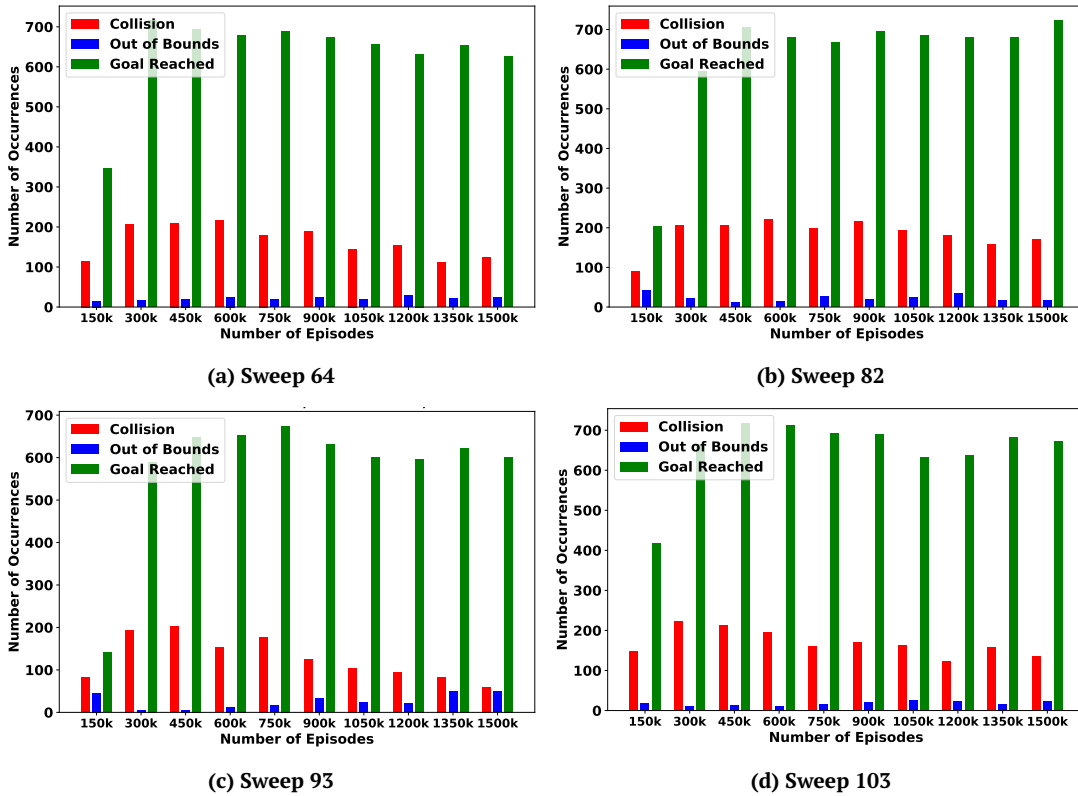


Figure 6.11: Training results of the 4 best-performing sweeps based on the number of times the agent collided, was out of bounds, and reached the goal. The figure shows statistics at every 150k checkpoint. It is seen that figure (c) outperforms the other figures in terms of collision with other obstacles.

The results show that the agent is capable of learning to avoid obstacles and to stay within its bounded area while still reaching its goal successfully. To determine which model to use between the 4 models, they will be compared to each other on the real setup in terms of oscillation and performance.

6.3 Behavior Engine

As mentioned previously in Chapter 2, the MPS setup is used for testing. In Fig. 6.12 the lab setup is illustrated.

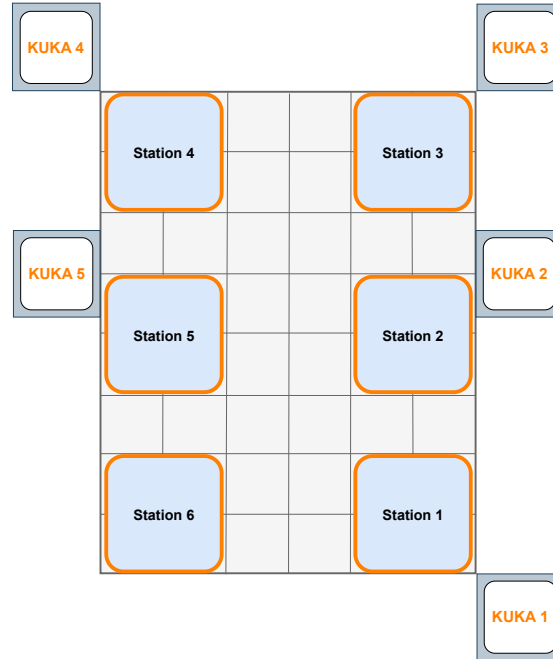


Figure 6.12: Illustration of an autonomous lab setup, which contains 6 boxed areas each representing a station, and 5 squares showing the placement of the robots.

The platform has areas for stations that contain hypothetical lab-related procedures in each, which will be the base of the sequences that the shuttles will be tested on. In Tab. 6.1 the stations are described according to the procedures and are only referenced in terms of their coordinates along with waiting time, which simulates time to perform the procedure, before the shuttle moves on to the next station.

Station no.	Coordinates (mm)		Procedure description	Waiting time
	X	Y		
1	600	120	Vision inspection	05 s
2	600	480	Dispensing powder	05 s
3	600	840	Stirring	10 s
4	120	840	Buffer	05 s
5	120	480	Dispense liquid	03 s
6	120	120	Manual station	05 s

Table 6.1: The station overview shows the procedures for each given station along with the wait time which simulates the procedure time when a shuttle arrives at the station.

A random sequence, shown in Tab. 6.2, was selected to show whether the behavior engine can be used to accomplish a task. In this sequence, 2 robots and 2 shuttles will be used. To do that, two parallel sequences will be created consisting of a parallel node along with two sequence nodes. Each shuttle is then moved in a separate sequence going to different stations.

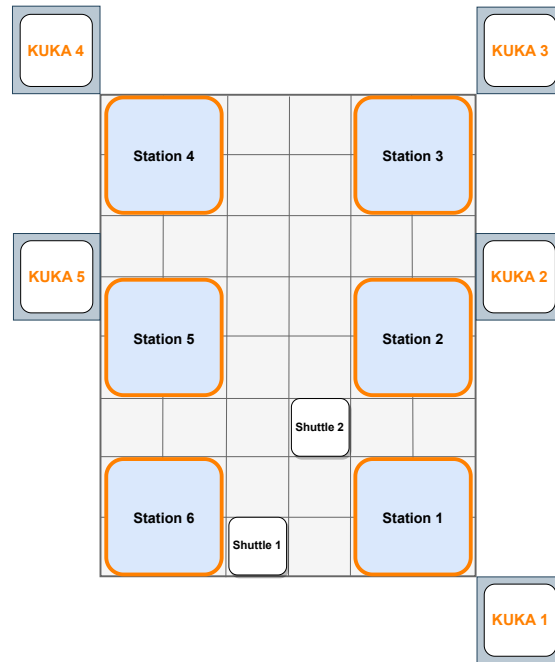


Figure 6.13: The area is divided into different stations, each having its own procedure. The figure illustrates the number of shuttles in the test case and their initial position.

SEQUENCES	
Shuttle 1	Shuttle 2
Station 1	Station 6
Station 2	Station 1
Station 3	Station 5
Station 6	Station 6

Table 6.2: Random sequence for 2 shuttles where each one will move to different stations from their starting position and ending in the same station.

A behavior tree is built and tested to see if this is feasible to accomplish. The test was a success where the shuttles moved correctly to their goal stations. Moreover, the parallel execution of nodes worked as expected, but due to one of the function calls blocking the calling thread, one of the shuttles waited for a longer time than expected. Furthermore, the KUKA robots at the designated stations succeeded to move to where the shuttles are. However, during the initial test, one of the robots collided with the camera stand. This is due to RoboDK not having the stand in its model. It also seems

that RoboDK selected a different configuration for the robot to go to the goal resulting in the collision.

One of the challenges of moving the shuttles to their target station is scheduling their movements. This is necessary as the shuttle would otherwise collide, therefore, the path each shuttle takes was carefully selected. The paths selected were always either moving along X then Y, or vice versa. A video recording of the performed test can be seen in this [link](#).

6.4 Deploying the RL agent

The test that follows is divided into two parts; an accuracy test and an obstacle avoidance test. The obstacle avoidance test is a three-part test, ranging from easy to difficult in level based on the placement of the obstacles, which must be avoided to reach the goal.

6.4.1 Accuracy Test

The motivation for this test is to assess the performance of the shuttle control of the trained RL models shown in Fig. 6.11. The test will be verify whether the shuttle can reach a given goal within an environment with no obstacles. These tests will be held up against the performance of the PMC of the same test case.

Test criteria

The test setup with the shuttle position and goal is seen in Fig. 6.14 followed by the criteria which the test should fulfill.

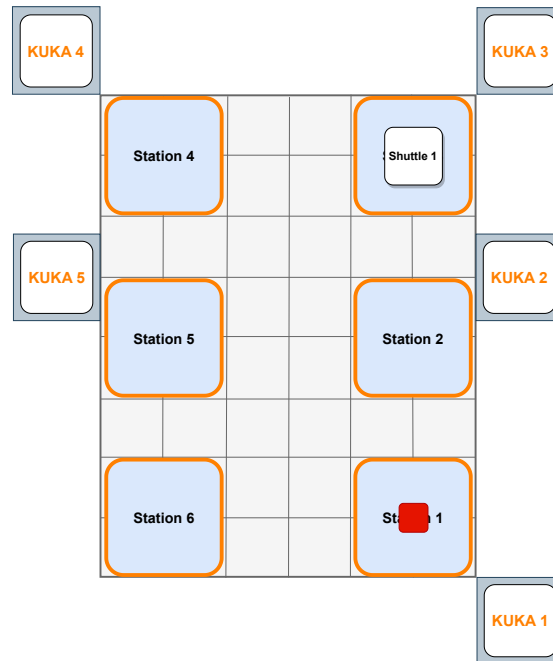


Figure 6.14: Figure illustrating the test case, where the shuttle has to move to from the initial position to the goal position which is marked in red.

- The test will consist of 10 trails for each RL model.
- The test will consist of the same goal and shuttle start position.
- The test will be timed and the average time for each agent will be determined.
- The deviation of the shuttle from the goal will be recorded and the mean determined.

Success criteria

The test case was performed using the PMC controller and it took the shuttle 1.293 seconds to reach the goal with no deviation from the set goal. This test result will be used as a baseline. The agents must fulfill the criteria stated below.

- Must reach the goal within 3 seconds.
- Must reach the goal with a maximum deviation of +/- 0.1 meters.

Results

The test data can be seen in Tab. 6.3, where the data shows that all controllers managed to get the shuttle to the goal on all counts during the tests. The table shows that the time to reach the goal vary between the 4 agents, with the fastest being RL-a82, with a mean time of 8.722 seconds, and the slowest being RL-a64, which took a mean time of 10.99 seconds. While RL-a82 may be the one to take the shortest time to the goal it is not the most accurate of the four. Although RL-a64 is the slowest of the agent to get reach the goal it is the most accurate one, with a mean deviation of 0.0359 meters, and the least accurate one being RL-a103 with a mean deviation of 0.540 meters. Based on the test results it is clear that the agents are significantly slower than the PMC and they exceed the time limit of 3 seconds. Therefore, the agents fail this criterion. When addressing the second criterion of accuracy the agents make up for its fallibility and succeed to stay within the deviation threshold, thus passing said criterion. The reason for the shuttle being slower when deploying the RL agents is due to the oscillation which makes the shuttle pace back and forth. This test shows the feasibility of using an RL agent for controlling the shuttles, although the oscillation must be alleviated. Based on the results, it was decided to use RL-a64 for the obstacle test, as it is the most accurate while still being able to avoid collision and avoid getting out of bound as seen in Fig. 6.11.

Controller	Mean t_goal	Max t_goal	Mean_dev	Max_dev	C1	C2
PMC	1.293 s	1.293 s	0	0	Passed	Passed
RL-a64	10.99 s	11.27 s	0.0359 m	0.0364 m	Failed	Passed
RL-a82	8.772 s	9.191 s	0.0467 m	0.0478 m	Failed	Passed
RL-a93	10.25 s	10.44 s	0.0515 m	0.0522 m	Failed	Passed
RL-a103	8.983 s	9.170 s	0.0540 m	0.0543 m	Failed	Passed

Table 6.3: PMC and RL agents test results. The results show that the RL agents are significantly slower compared to the PMC, although they reach the goal, staying within the threshold.

6.4.2 Obstacle Avoidance Test

The purpose of this test is to verify and assess the agent’s obstacle-avoidance capabilities. The test consists of controlling one shuttle to reach a given goal, while the remaining shuttles will serve as obstacles in the environment. The obstacle avoidance test is divided into three levels, easy, intermediate, and difficult. The level of complexity is based on the placement of the shuttles and the goal position. The test will have simple PASS or FAIL criteria based on whether the shuttle reaches the goal or collides during its course. The shuttle placement is specified under each test case, along with the test results.

Easy

The placement of the shuttles, seen in Fig. 6.15, are contained in the middle of the platform to make it an easy scenario. This scenario allows the for the shuttle to surpass the obstacles to reach the goal in the upper right corner, as multiple paths can be taken to arrive at the goal. The test was performed 10 times to determine the success rate of the agent in this given scenario. A video of one of these test runs can be seen via this [link](#).

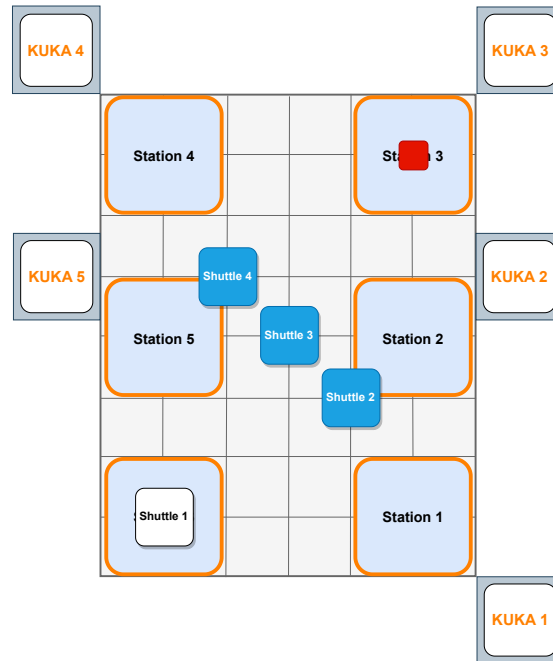


Figure 6.15: Illustration of the test case showing the placement of the obstacles along with the start position of the shuttle and the goal position marked in red.

The test video shows that the agent manages to reach the goal. The agent succeeds in all 10 trials. According to the recorded test data, it reaches the goal with a mean time of 16.57 seconds with a mean deviation of 0.0313 meters to the goal. Although the maneuvering of the shuttle oscillates it completes the test showing that it can operate in the current environment confidently, proving that it is a functional model.

Intermediate

In this test, the obstacles are arranged similarly to the previous one but further apart increasing the difficulty in reaching the goal. The initial position of the shuttle and the goal can be seen in Fig. 6.16. This scenario requires the agent to either go all the way around shuttle 4 (top obstacle) and to the goal or to go straight down towards the goal, but where is more difficult to get past shuttle 2 due to the narrow space. This test case consisted of 10 trials.

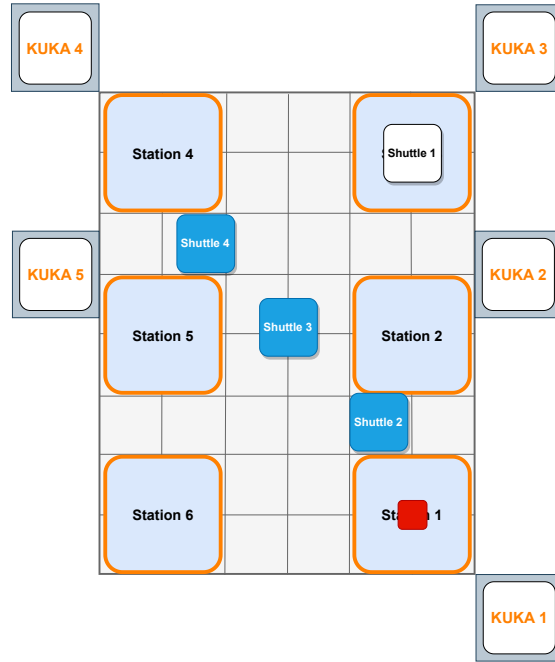


Figure 6.16: Illustration showing the obstacle placement for the intermediate test with the initial position of the shuttle and the goal position.

As anticipated the agent collides with shuttle 2 (bottom obstacles) as it attempts to move alongside it to get to the goal. During the test, the agent fails all 10 trials and collides each time at the same position. This is due to the oscillation which makes it challenging for the shuttle to move through narrow areas, making it collide with an obstacle when being side-by-side with it. This behavior can be seen in this [link](#) for the test video. It was decided to try and shift shuttle 2 to the left by 15mm to see if the agent would be capable of reaching the goal with a bit more space to maneuver. As expected the agent reached the goal in this particular trial which can be seen in this [link](#).

Difficult

For the last test, it was decided to arrange the obstacles in another configuration which increases the difficulty significantly. In this scenario, the shuttle only has one way to reach the goal compared to the previous two cases. Here, the goal position is surrounded by the obstacles, partially shielded, as can be seen in Fig. 6.17. Although the "intermediate" test failed it was decided to proceed with the "difficult" test to see how the agent will navigate in situations such as these. This will help to further analyze the agent model and look for any behavior tendencies. The test is performed 10 times as the preceding ones.

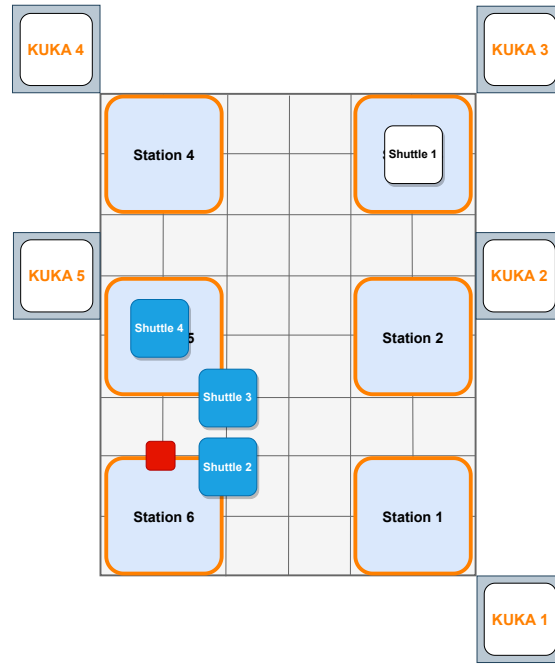


Figure 6.17: Illustration showing the obstacle placement for the difficult test with the initial position of the shuttle along with the goal position.

During the test it could be observed that the agent moves down effortlessly until it reaches shuttle 2 (obstacle) where it collides with it in an attempt to navigate around it. A recording of this can be seen in this [link](#). This happens in all 10 trials performed, making the agent fail this test.

What is noteworthy in these 3 test cases is the observation of the agent's behavior. It was noticed that the agent tends to move along the sides, meaning that the shuttle would always seek to go around the obstacles as opposed to going in between them. Furthermore, it was observed that the agent moves in close proximity to the obstacles when moving around them to avoid them, which made it prone to colliding with the obstacles. These behavior tendencies create issues when trying to reach a goal in scenarios such as the one portrayed in the "difficult" test case. In the mentioned test the agent only had one path to reach the goal as it could not go around them all to get to the goal, which forces it to go towards the obstacles to get to the goal. Due to its ten-

dency of moving very close to the obstacles, it ends up colliding with them instead. These behaviors of the agent have led to the belief that it would struggle in environments where there are many obstacles, as there is a high probability that it will be unable to navigate in between due to its preference for moving around obstacles.

Chapter 7

Discussion

This chapter presents the evaluation of the solution implemented in this project, highlighting its strengths, weaknesses, and overall effectiveness, to determine if it meets the desired objectives stated in Chapter 4. Furthermore, to mitigate the issues uncovered during testing, it is decided to explore areas of the solution that can be improved.

7.1 Evaluation of the Solution

The solution presented in this project consists of 2 major components, a SBS referred to as the behavior engine and the RL agent which both are tested out on a physical setup, the MPS, as a proof of concept to show the feasibility of the solution. The following evaluation is used to validate the capabilities of the solution and to target areas in need of improvement, which will be discussed in the section after.

7.1.1 Behavior Engine

The evaluation will be based on the objectives previously stated for the Behavior engine. Each objective will be discussed to determine if the system fulfills them.

Easy programming of lab task

One of the main objectives for this project was to develop a system that provides the lab workers with a user-friendly system allowing visual programming of robots for setting up experiments. In addition to this it was desired to follow a SBS approach by also implementing a skill library with relevant lab-related skills. The system implemented achieved these goals by creating a SBS by using behavior trees. Furthermore, a user interface was created that allowed for visual programming in the form of drag-and-drop of skill nodes to create tasks. The skill library implemented consists of skills such as stirring, dispensing, color detection, and moving. In addition to this, the user could also add wait nodes, inverters, and other similar functions to plan out their task. The overall use of the behavior engine was tested out by creating a BT with a task for two shuttles with different sequences. This was done to see the capabilities of the BT as well as determine how manageable it is to use the interface, based on intuition. Despite the test proving the fulfillment of this objective, it can be argued that in regards to determining the user-friendliness of the interface, it is difficult to express an unbiased opinion due to the authors of this project having crucial insight into the system's functionality and prior knowledge of BTs. Therefore it can be concluded that a survey with non-technical individuals (preferably lab workers) must be conducted to properly validate the benefits of the solution.

Tasks in parallel

As mentioned in Section 2.2, to accelerate experiments for material discovery, it would be beneficial to be able to run experiments in parallel, which would cut down the experiment time significantly. Therefore, it was decided to aim for a system that allows for creating parallel skill sequences to run multiple experimental procedures simultaneously. This is successfully implemented in the project to a reasonable extent. The behavior engine makes it possible to run things in parallel. However, due to the implementation being asynchronous, this limits what can be run in parallel. This can cause unreal expectations of the system behavior if the calling thread will get blocked. Therefore, it should be made clear which actions and conditions can be run asynchronously allowing the user to utilize the system capabilities to their full potential.

Feedback

As described in Appendix A.1, the user must have feedback from the User Interface (UI) to follow what is going on in the system. With no feedback, the user may feel confused or unsure of the proper use of the system. It was therefore decided that one of the objectives for the behavior engine is to have some feedback to the user. One of the more crucial feedback that was implemented is the visual active color change during runtime. When running a BT the colors changes according to its status. Inactive nodes are blue, nodes that are running becomes yellow, completed (succeeded) nodes are green, and nodes that fail turn red, which can be seen in Fig. 5.3. This gives the user an indication of which node is currently running and where the BT stopped in case of an error, making it easy to debug or make changes in the tree.

7.1.2 Reinforcement Learning Agent

The evaluation will be based on the objectives previously stated for the RL agent. Each objective will be discussed to determine if the agent fulfills them and to which extent. Furthermore, the performance of the agent will be discussed in comparison to the existing PMC, to determine the overall feasibility of the agent controller.

Shuttle control

This objective was set to create an RL model to control the shuttle to move to a goal within the reachable space. This model should be deployable on any shuttle to achieve control of a multi-robot system. Following the implementation of the RL agent the model was tested, without obstacles along its path, on the MPS to show whether it can be used to reach a given goal. The agent was assessed based on time and accuracy. A tolerance of 0.1 meters was set to ensure that the agent can reach the goal and does not stray too far from it. The shuttle needs to be accurate for it to be reachable by the robot assigned to the given station. The test results show that the agent is capable of commanding the shuttle to the given position with a mean deviation of 0.0359 meters, fulfilling the set test criterion. However, due to oscillation, the shuttle took a mean time of 10.99 seconds, which is significantly higher than the PMC, which took 1.293 seconds.

It is suspected that the oscillation is caused by the integration of the actions given from the RL model, which has been routed through the PMC. The RL model outputs the ac-

tion as a vector of 2 dimensions representing linear velocity in x and y. These velocities were integrated into a position and the magnitude of the velocity vector was given as an end speed in the PMC. This means that updating the position must happen at a fixed rate ensuring that the PMC receives another position update before the shuttles reaches the current given position.

Collision avoidance

The objective of collision avoidance was set due to the desire of having multiple shuttles operating in the same space without colliding or going out of bounds. The obstacle avoidance test in Chapter 6 shows that the trained agent managed to avoid all obstacles in some scenarios while it lacked in others. The repeated collisions occurred in the intermediate and difficult tests. In these cases, the shuttle attempts to move closer to the goal even if a collision might occur. This can be caused by not rewarding the agent based on the best strategy to follow. Moreover, the tuning of the parameters could have included metrics such as the number of collisions, out-of-bounds, and goals reached to optimize the parameters. Therefore, the collision avoidance capabilities must be improved before the model is usable in a real system.

Physical setup

The objective was to be able to train any model which was transferable to the real MPS. This was achieved by training the model with `stable_baselines3` and exporting it to an ONNX format that can be used in combination with the behavior engine. Although the solution is implemented with scalability in mind, it should be noted that deployment of the model is only possible on systems identical to the one trained on. This means that the number of shuttles has to be identical making the current implementation not scalable or adaptable without further training of the model.

7.2 Improvements

When implementing the solution, many decisions were made shaping the solution into its current version. Naturally, when making choices, it is inherent that selecting certain things automatically means the deselection of others. This section covers the reflection of the implemented solution. It dives into the features and improvements that, in hindsight, would have been a great addition to the solution and what could have been done differently.

7.2.1 Behavior Engine

As previously mentioned the purpose of the behavior engine is to enable lab-workers to effortlessly create robot tasks for experiments. This was kept in mind when designing and implementing the system, which lead to the use of behavior trees and more graphic-based programming, to keep the creation of tasks at a higher level. Although the system seems intuitive, it can be argued whether it caters well to the intended user. This is because it is still required of the user to have some understanding or prior knowledge of behavior trees to make use of them. This may make the non-technical user refrain from using the system. One way to help the user with this is to have the interface

provide hints and interactive messages to guide them to create tree designs to accomplish different tasks. Another feature that could have been added was the possibility to customize the behavior engine, allowing the user to name node types (actions, root, decorators, etc.) according to their theme, which would make the experience and use of the interface more intuitive for the user. During the test of the behavior engine, it was noticed that even small tasks could require large trees, increasing the complexity. One way to avoid this, and to ensure that the structure of the tree is comprehensible for the user, would be to incorporate the use of subtrees.

7.2.2 Reinforcement Learning Agent

Even though the agent can reach its goal while avoiding obstacles, the agent still commands the shuttle to move closer to the goal even if it will cause a collision. This should be mitigated by rewarding the agent based on the best strategy it should use instead of always rewarding it based on reaching the goal. Furthermore, the reward weights can be tuned using Weights and Biases along with the other hyperparameters by minimizing the number of collisions and maximizing the number of times to reach the goal. Another limitation of the current implementation is the observation space which can only observe 3 shuttles. This limits the generalization ability of the model, requiring retraining if the need arises to deploy more shuttles. Therefore, alternative approaches must be investigated to determine how this observation can be given to the agent.

Chapter 8

Conclusion

The increase in the deployment of robots in factories has led to the need for alternative programming approaches that are more cost-efficient and do not require expert knowledge. It has therefore been the motivation behind this project to develop a solution for easy control and creation of robot tasks. The approach presented in this project offers several advantages over traditional programming methods. A system with intuitive and visual programming utilizing both BT and RL for control has been implemented making it more accessible for the non-technical user. The scope of this project was focused on the use of such a system in the context of MAPs, more specifically for the orchestration of self-driving labs. To validate the full solution pipeline it was decided to focus on a use case tailored to the MPS, which is a multi-robot system, consisting of shuttles and robot manipulators. The research done in regards to self-driving labs uncovered the common challenges in automating lab procedures. This included the complexity of transferring tasks, manually performed by humans to robots, due to differences in dexterity, as well as the current layout of labs not providing the means for automating experiments. It was concluded that to support MAPs it was important to re-invent the design of today's lab environment and provide a system that could run experiments in parallel and be easy to program and reconfigure. This lead to the following problem statement.

How can planning and execution of lab-related tasks be done using BTs in the MPS system enabling easy programming and system reconfiguration, while achieving control of the shuttles with the help of an RL agent?

With this in mind, a behavior engine was implemented using the principles of BTs for high-level task planning with the integration of a RL agent for the use of control and collision avoidance for the shuttles. The tests included in Chapter 6 prove the feasibility of the solution, as the creation and execution of a BT through the behavior engine are accomplished, making it a viable solution for robot task planning. Furthermore, it was shown that the trained RL model is capable of navigating around the environment to get to the desired goal, when there are no obstacles. However, the RL agent seems to lack when meeting many obstacles on its course to the goal. This was due to the oscillation and behavior tendencies of the model. Despite this, the agent still shows potential and may be suitable for self-driving labs by improving the reward system and performing more vigorous training to ensure a robust model. In general, the system acts as a proof of concept showing that it is possible to implement such generalized system, but has more room for improvement before it can be presented as a potential solution. In conclusion, our novel approach combining BT and RL for easy programming of multi-robot systems in MAPs shows promising results. The advancements made in this project contribute to the overall goal of automating lab processes leading to the acceleration of material discovery.

Chapter 9

Future Work

This chapter contains the relevant future work that the solution could benefit from, focusing on aspects such as increasing the usability and overall effectiveness of the system.

Preview of the Robot's Movement

The features of the behavior engine allow the user to create complex tasks and execute them on the real setup. However, the lack of visualization options to preview the robot's movement before execution makes designing the task more challenging. Therefore, adding the option to view the current state of the robot and the planned behavior would make it more suitable for deployment.

Behavior Tree Assistant

To assist the user with designing the tree, a Large Language Model (LLM) can be used, which offers more capabilities than the ChatBot used in ChemOS. The integration of an LLM would allow the user to query the system about how to accomplish a specific task and learn more about behavior trees and their capabilities. This can be done by providing the language model with the system capabilities along with which skills are implemented in addition to their description and ports.

Automatic Behavior Tree Creation

Using an LLM, it would be possible to create the tree automatically and give the user the ability to edit it before executing it, an example of this can be seen in [52]. Other algorithms also exist which can assist to create a behavior tree automatically such as the Planning and Acting using Behavior Trees (PA-BT) approach [24]. This algorithm uses an iterative approach to build the tree by starting with the goal and then moving backward to find a sequence to accomplish the goal.

Global Path Planning

Currently, the reinforcement learning agent acts as a local planner and requires global planning to find the path to reach its goal. The use of a global planner is needed when the environment is more complicated or when the goal is unreachable.

Task Scheduling

It was observed during the testing on the physical setup that scheduling the movement of the shuttles to accomplish a specific sequence becomes more complicated with the increase of the number of shuttles. The use of a task scheduler ensures that no deadlock occurs in the system, where two shuttles would wait for each other to move.

Hardware Configuration

The implemented skills work only on the current setup and adding hardware requires changing the skills themselves. Therefore, adding the option to configure the hardware from the frontend makes it easier for the user.

ROS support

Supporting ROS is beneficial due to the wide array of features and robots supported by it. This includes mobile robots' and manipulators' drivers as well as packages used for perception, navigation, control and visualization. Furthermore, a variety of grippers and sensors are integrated with it, making ROS beneficial for hardware abstraction.

Bibliography

- [1] International federation of Robotics. *World Robotics Report: “All-Time High” with Half a Million Robots Installed in one Year*. URL: <https://ifr.org/ifr-press-releases/news/wr-report-all-time-high-with-half-a-million-robots-installed>.
- [2] Congyu Zhang Sprenger and Thomas Ribeaud. “Robotic Process Automation with Ontology-enabled Skill-based Robot Task Model and Notation (RTMN)”. In: *2022 2nd International Conference on Robotics, Automation and Artificial Intelligence (RAAI)*. 2022, pp. 15–20. DOI: 10.1109/RAAI56146.2022.10092996.
- [3] Martha M. Flores-Leonar et al. “Materials Acceleration Platforms: On the way to autonomous experimentation”. eng. In: *Current opinion in green and sustainable chemistry* 25 (2020), pp. 100370–. ISSN: 2452-2236.
- [4] Toronto University. *Accelerating the discovery of materials and molecules needed for a sustainable future*. URL: <https://acceleration.utoronto.ca/>.
- [5] DTU Capex. *PIONEERING HOW SUSTAINABLE MATERIALS FOR POWER-TO-X ARE INVENTED*. URL: <https://www.dtu.dk/capex>.
- [6] AAU. *Smart production - Aalborg University*. URL: <https://www.mp.aau.dk/research/research-areas/robotics-and-automation/smart-production>.
- [7] B&R Industrial Automation. *B&R heralds the beginning of multidimensional manufacturing with ACOPOS 6D*. URL: <https://www.br-automation.com/en/products/mechatronic-systems/acopos-6d/>.
- [8] Planar Motor Incorporated. *Planar Motor System*. URL: <https://www.planarmotor.com/en/products>.
- [9] Beckhoff Automation. *XPlanar | Planar motor system*. URL: <https://www.beckhoff.com/en-en/products/motion/xplanar-planar-motor-system/>.
- [10] RoboDK. *RoboDK driver for KUKA*. URL: <https://robodk.com/doc/en/Robots-KUKA-RoboDK-driver-KUKA.html>.
- [11] Sarvanan Chidambaram - Vikas Sajjan. *How Heterogeneous Systems Evolved and the Challenges, Going Forward*. URL: <https://www.mp.aau.dk/research/research-areas/robotics-and-automation/smart-production>.
- [12] “Autonomous Chemical Experiments: Challenges and Perspectives on Establishing a Self-Driving Lab”. eng. In: *Accounts of chemical research* 55.17 (2022), pp. 2454–2466. ISSN: 0001-4842.
- [13] Loic M. Roch et al. “ChemOS: An orchestration software to democratize autonomous discovery”. eng. In: *PloS one* 15.4 (2020), e0229862–e0229862. ISSN: 1932-6203.

- [14] Benjamin Burger et al. “A mobile robotic chemist”. eng. In: *Nature (London)* 583.7815 (2020), pp. 237–241. ISSN: 0028-0836.
- [15] Chibum Lee Jaewan Choi Geonhee Lee. “Reinforcement learning-based dynamic obstacle avoidance and integration of path planning”. In: (2021). DOI: <https://doi.org/10.1007/s11370-021-00387-2>.
- [16] Mikkel Rath Pedersen et al. “Robot skills for manufacturing: From concept to industrial deployment”. In: *Robotics and computer-integrated manufacturing* 37 (2016), pp. 282–291. ISSN: 0736-5845.
- [17] Casper Schou et al. “Skill-based instruction of collaborative robots in industrial settings”. In: *Robotics and computer-integrated manufacturing* 53 (2018), pp. 72–80. ISSN: 0736-5845.
- [18] Rasmus Eckholdt Andersen et al. *Integration of a Skill-based Collaborative Mobile Robot in a Smart Cyber-Physical Environment*. eng. 2017.
- [19] Matthias Mayr et al. “Learning of Parameters in Behavior Trees for Movement Skills”. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 7572–7579. ISBN: 1665417145.
- [20] Ready Robotics. *ForgeOS*. URL: <https://www.ready-robotics.com/solutions/upskill-your-people>.
- [21] Ready Robotics. *Documentation*. URL: <https://www.ready-robotics.com/support/documentation>.
- [22] Picknik. *Picknik homepage*. URL: <https://picknik.ai/studio/>.
- [23] RiACT. *RiACT homepage*. URL: <https://www.riact.eu/>.
- [24] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI: An Introduction*. eng. Ithaca: Cornell University Library, arXiv.org, 2018.
- [25] Aske Plaat. *Deep Reinforcement Learning*. eng. Singapore: Springer, 2022. ISBN: 9789811906374.
- [26] Google Deepmind. *DeepMind x UCL RL Lecture Series - Introduction to Reinforcement Learning [1/13]*. URL: <https://www.youtube.com/watch?v=TCCjZe0y4Qc>.
- [27] Jingles (Hong Jing). *Reinforcement Learning — The Value Function*. Ed. by towards science. URL: <https://towardsdatascience.com/reinforcement-learning-value-function-57b04e911152>.
- [28] Renu Khandelwal). *Reinforcement Learning: On Policy and Off Policy*. URL: <https://arshren.medium.com/reinforcement-learning-on-policy-and-off-policy-5587dd5417e1>.
- [29] (Hugging face courses). *Introducing the Clipped Surrogate Objective Function*. URL: <https://huggingface.co/learn/deep-rl-course/unit8/clipped-surrogate-objective?fw=pt>.
- [30] John Schulman et al. “Proximal Policy Optimization Algorithms”. eng. In: (2017).

- [31] Unity technologies. *Training Configuration File*. URL: <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md>.
- [32] Stablebaseline3. *PPO*. URL: <https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html>.
- [33] AurelianTactics. *PPO Hyperparameters and Ranges*. URL: <https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe>.
- [34] Open Robotics. *What is Isaac Sim? – Omniverse Robotics Documentation*. URL: https://docs.omniverse.nvidia.com/app_isaacsim/app_isaacsim/overview.html.
- [35] Deepmind. *Overview - MuJoCo Documentation*. URL: <https://mujoco.readthedocs.io/en/stable/overview.html>.
- [36] Cyberbotics. *Webots documentation - Introduction to Webots*. URL: <https://cyberbotics.com/doc/guide/introduction-to-webots>.
- [37] Open Source Robotics Foundation. *Gazebo*. URL: <https://classic.gazebosim.org/>.
- [38] Open Robotics. *Gazebo - Docs: Get Started*. URL: <https://gazebosim.org/docs>.
- [39] Marian Körber et al. “Comparing Popular Simulation Environments in the Scope of Robotics and Reinforcement Learning”. eng. In: (2021).
- [40] Microsoft. *Architectural principles*. URL: <https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles>.
- [41] Microsoft. *Get started with Entity Framework 6*. URL: <https://learn.microsoft.com/en-us/ef/ef6/get-started>.
- [42] GraphQL. *Frequently Asked Questions (FAQ)*. URL: <https://graphql.org/faq/#does-graphql-use-http>.
- [43] Microsoft. *Common client-side web technologies*. URL: <https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-client-side-web-technologies>.
- [44] Vue.js. *Introduction*. URL: <https://vuejs.org/guide/introduction.html>.
- [45] OpenJS Foundation. *Build cross-platform desktop apps with JavaScript, HTML, and CSS*. URL: <https://www.electronjs.org/>.
- [46] Ionic. *Capacitor: Cross-platform Native Runtime for Web Apps*. URL: <https://capacitorjs.com/docs>.
- [47] OPC Foundation. *Unified Architecture*. URL: <https://opcfoundation.org/about/opc-technologies/opc-ua/>.
- [48] Association for Advancing Automation (A3). *GigE Vision*. URL: <https://www.automate.org/a3-content/vision-standards-gige-vision>.

- [49] Wikipedia. *HSV color solid cylinder*. URL: https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder.png.
- [50] Packt. *Object detection using color in HSV*. URL: <https://subscription.packtpub.com/book/data/9781789537147/1/ch01lvl1sec09/object-detection-using-color-in-hsv>.
- [51] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: <http://jmlr.org/papers/v22/20-1364.html>.
- [52] Yue Cao and C. S. George Lee. "Robot Behavior-Tree-Based Task Generation with Large Language Models". eng. In: (2023).
- [53] Ivan Schneiders. *Bad design: Deconstructing the Norman door*. URL: <https://ivanschneiders.medium.com/bad-design-deconstructing-the-norman-door-4420fd84b960>.
- [54] Anton Nikolov. *Design principle: Consistency*. URL: <https://uxdesign.cc/design-principle-consistency-6b0cf7e7339f>.
- [55] RILEY ROTH. *3 Key Elements for Great UX Design: Affordances, Signifiers, and Feedback*. URL: <https://careerfoundry.com/en/blog/ux-design/affordances-signifiers-feedback/>.
- [56] Pratyush Pandab. "Ingredients of Good Design: Affordance, Emotion and Complexity". In: (May 2013).
- [57] Zeerek Ahmad. *State Machines vs Behavior Trees: designing a decision-making architecture for robotics*. URL: <https://www.polymathrobotics.com/blog/state-machines-vs-behavior-trees>.
- [58] Priyam basu. *Part 4 — Exploration and Exploitation*. URL: <https://medium.com/iecse-hashtag/rl-part-4-exploration-and-exploitation-859bc294e2b0>.
- [59] Beckhoff Automation. *ADS-Communication*. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/cx8190_hw/5091854987.html.

Appendix A

Appendix

A.1 What is a good user interface?

It should be emphasized that the importance of a good user interface is paramount, as the solution aims to make the programming of robot tasks easy and more intuitive for the user. To achieve this it is relevant to understand the intricacies involved in designing an UI. UI design is a whole field on its own and consists of much more than creating the visuals of a product. When determining the best way for the user to interact with a product and how the message of said item is conveyed to the user there is one other element that plays a major role, that is User Experience (UX). UI - and UX design go hand-in-hand, as both are equally important for the user to understand and benefit from the solution as intended.

In simple terms, UI embodies all the physical attributes, such as buttons, size, icons, and colors, which enables the user to interact with the product. While UX the users interact with the product as a whole, including how they feel when using it. It is possible to have a fully functional product based on UI design but without considering what good UX is, it is possible to leave the user feeling confused and unhappy with their experience. A widely known example of this is the concept of the Norman door, named after Don Norman who is considered to be the founder of UX. It is a term used for any door that is confusing to use and is a concept that can be transferred to other objects when considering the design and user experience [53].

In Fig. A.1 two doors are shown, both of which may lead the user to be unsatisfied. When ignoring the signs on the door and only looking at the doors themselves, they may be counter-intuitive. The door to the right has the handle shape that is commonly associated with a "pull" door, and the one to the left looks like a "push" door. This may lead the user to operate the door incorrectly based on their intuition. Although the doors themselves are fully functional the design can lead to the user having a negative experience. In the example above the doors are equipped with signs that work as indicators to help the user, but in general good UI is considered to be easy and intuitive [53].

To further understand these two principles (UI and UX) and how they impact the solution, the following sections will dive into both, to derive the knowledge needed for the implementation phase.

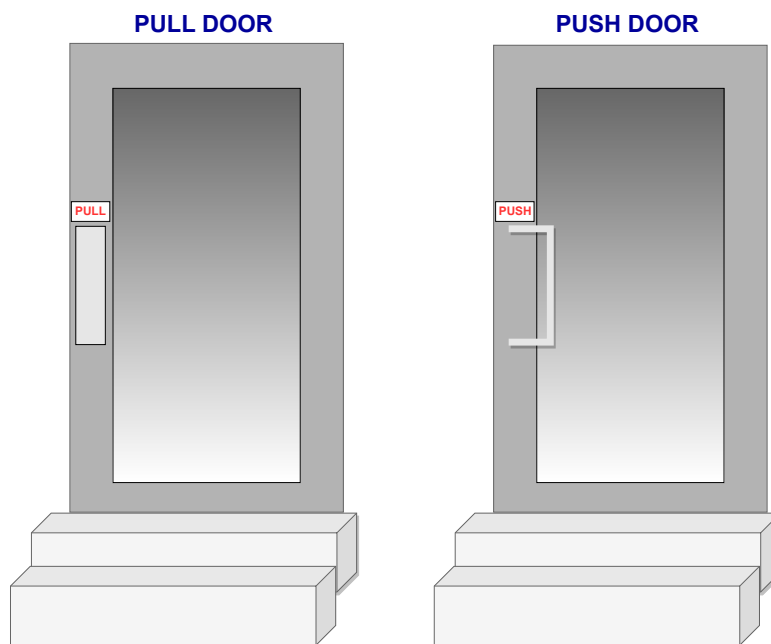


Figure A.1: Example of two doors demonstrating the 'Norman door' concept regarding how bad UX design can lead to confusion and unsatisfactory experience for users.

A.1.1 User Interface design

When designing a UI, consistency is key, as consistency leads to more intuitive design. This is due to the element of familiarity, which makes it easier for the user to learn. Humans naturally learn by looking for patterns so staying consistent throughout (menus, buttons, etc.) increases the learnability, due to the predictability of the interface [54]. There are four types of consistency:

Visual consistency

Being consistent with the visual components across the entire platform, such as buttons, fonts, labeling, and colors. This increases the learnability of the interface and provides less distraction for the user[54].

Functional consistency

With functional consistency it is important to keep the control functions such as return buttons, slider, toggle buttons, and general control flow in the system should function the same way throughout the platform. In other means of it looks like a button it should act like one too. This will increase the predictability making the user feel safe and confident with the interface [54].

Internal consistency

This term refers to the previous two as a combination. Which improves the usability of the interface, making it easier for the user to get introduced to new features [54].

External consistency This term refers to when the internal consistency is extended

to more or all products. This makes it easier for the user to transfer their acquired knowledge to a new product. This eliminates the tediousness of learning an interface all over again[54].

A.1.2 User Experience design

When referring to good UX it can be described as a product having affordance. Affordance can be used as a way to qualify how well a product can communicate its function. A Product with good affordance will make the user know right away what it does and how to use it, without the user having to think about it. This is due to the product feeling natural and following logical associations. There are simple ways to enhance the affordance of a product. This can be done by using signifiers and feedbacks [55][56].

Signifiers and feedbacks

Signifiers are used as indicators to let the user know what affordance the product has. Affordance should not be understood as a physical property that a product possesses but can be described as the relationship between the product and the user. Signifiers are also known as perceived affordance, defined by Don Norman according to [56]. There are many simple examples of affordance and signifiers found in everyday items, such as a thermo-mug that has a thumb-shaped button, which is affordance as the user immediately knows that it can be pressed based on the shape of it, and the lock/unlock icons on it usually serves as indicators, telling the user that it is possible to lock the mug spout. In the digital world, signifiers are more important as there is no physical feel to the interface. On webpages or other interfaces, it is commonly seen that when you hover over a button with the mouse it usually changes in color, this is considered a signifier as it indicates to the user that it is possible to click on the button. In other cases, the mouse arrow becomes a pointer indicating that it is a button, or when clicking on a text field the arrow becomes a cursor, letting the user know that it is possible to type here.

Another important thing related to signifiers and affordance is feedback. A simple example of feedback is the small LED found on a monitor, the LED itself is a signifier, and the color it emits is the feedback, as the user will know if the monitor is powered or not based on the light. Feedback is a very important element to increase affordance. It is important that when a user interacts with an interface they always receive some feedback when performing an action, this makes the user know what is going on and avoids confusion. This is why feedback should be given even in case of an error or an unsuccessful action. Imagine logging into an email account and typing the wrong password but not receiving a message notifying you that the password was wrong. This creates confusion and uncertainty in the user, making them have a negative experience. According to psychologist Alice Isen in [56], users tend to overlook flaws and minor inconveniences when being in a good mood but have more attention to details when being stressed or unhappy. This has a big impact on how the user perceives a product or a UI, as a small error when being relaxed and happy may go unnoticed, but when being in a stressful situation the same error may feel like a major inconvenience and

make the experience upsetting and hinder the learnability of the product [56].

As explained in the above sections, Both good design for UI and UX is vital for having an interface that the user will be able to benefit from. As the user interface will be concerning making skill-based programming easy and comprehensible for non-technical users it will be beneficial to follow the principles explained above. This will ensure that the UI is intuitive and the user will be able to focus on learning how to program tasks and succeeding at it. In Chapter 5, all decisions and considerations in regard to design choices will be described in depth.

A.2 Finite State Machines

In automata theory, FSMs are defined as a finite number of states that are in a predetermined sequence that run automatically. FSMs are a mathematical modeling tool that describes the relationship between different states and their transitions. States describe the system status whereas a transition is the action to take. Transition happens either by checking if a condition is met or by an event that triggers the action. This type of system has gained popularity in engineering to describe system behaviors. FSMs are categorized into deterministic, where every state has only one possible transition, and non-deterministic where states can have one or more transitions or no transition at all. FSMs are divided into 4 different classes; acceptors, classifiers, transducers and sequencers.

Acceptor FSMs

Acceptors are used to transition between states producing only two outputs, accepting or rejecting, or more formally a binary output.

Classifier FSMs

Classifiers are similar to acceptors but have a strict condition for having more than 2 outputs.

Transducer FSMs

Transducers are most commonly used in control applications, defined by two different types; Moore machine and Mealy machine. In Moore machines, the output is only dependent on the state and gets triggered by an entry action. Mealy machines are similar to Moore machines in that they depend on the state, but they also depend on inputs.

Sequencer FSMs

A special case of classifiers or transducers. The states in a sequencer FSM do not change thus generating a fixed sequence. As previously stated, Moore and Mealy's machines are commonly used for control architecture. These machines have a few advantages:

- Intuitive and straightforward

- Simple to implement
- Well-known and widely used in the industry

Despite those advantages of FSMs, there are some drawbacks when the model's complexity increases. Some of these drawbacks are listed below:

- Susceptible to errors when adding or removing states, since it might require re-evaluating the whole state machine
- Difficult to scale when the system grows in complexity and the number of transitions increases
- Re-purposing existing FSMs is impractical when transitions depend on internal variables

To mitigate some of these drawbacks, hierarchical FSMs (HFSMs) were developed. A concept of a superstate was defined in HFSMs, which is a state that has multiple sub-states allowing for a more complex and modular representation of a system. In HFSMs, a parent state can have multiple child states, where the transition can be defined at both the child and parent levels.

FSM versus BT

As both FSM and BT are described, the advantage and disadvantages can be discussed, to determine which is preferable for the solution implemented in this project. For simple systems with few states needed FSM is ideal as it is efficient and rigid, but as soon as the need for many states occurs, FSM can become complex and incomprehensible due to its many transitions and conditions. When dealing with robotics tasks with a large amount of decision-making, requiring complex behaviors, then BT is considered to be more beneficial to use. BT differs from FSM as a 'tick' is propagated through the tree in the order determined by the control nodes. BTs do not have explicit states as FSM and are therefore also more agile and easy to add changes to. Furthermore, BTs are easier when it comes to error handling as the tree is more comprehensible as the execution order of the nodes is determined based on the control nodes.

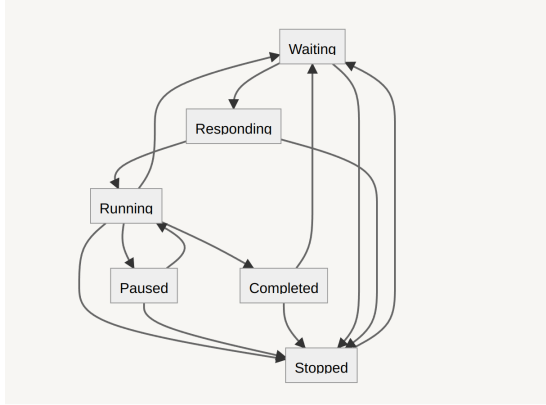
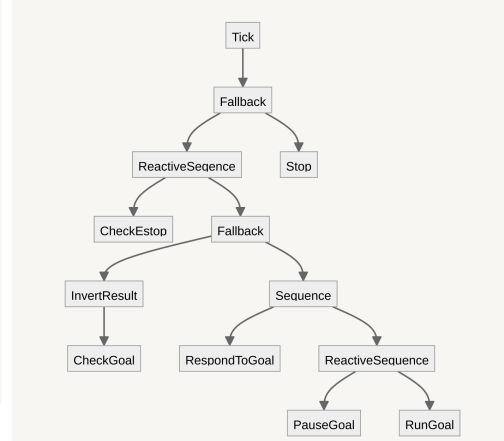
Finite State Machine**Behavior Tree**

Figure A.2: The figures show an example of a finite-state machine and a behavior tree side-by-side to show the difference in characteristics of both structures. [57]

In the Fig. A.2 an example is shown of the same type of task implemented as both FSM and BTs. As can be seen FSMs, quickly tend to become unorganized and hard to comprehend with all the transition connections between the different states, while as for the BTs, the structure stays simple due to the tree-like structure, making it easy to follow regardless of the complexity of the task. In the following table A.1 the characteristics of FSM and BT are summarized to get an overview of the advantages and disadvantages of the two, to determine which should be used for the project.

Finite State Machines	Behavior Trees
- Modular	- Modular
- Rigid	- Flexible
- Simple	- Complex
- Require many states	- Good for debugging
- Explicit states and conditions	- No need for explicit states
	- Error handling (fallback)
	- More transparent

Table A.1: Comparing the characteristics of FSM and BT to provide an overview of their advantage and disadvantage in regards to the solution.

As stated in the table, BT provides a more flexible and agile system, which is needed for easy robot task creation, and gives the transparency during runtime that is needed for debugging. Due to these properties, it was decided to use BT for the skill-based system which is to be implemented in the solution for this project.

A.3 Exploration and Exploitation

For optimized learning, it is important to have a balance between exploration and exploitation. Exploration is when the agent explores the environment, while exploitation is when the agent uses the information obtained from the environment. During the initial state of training an agent usually benefits from exploring as the environment remains unknown, the more observations the agent receives it will be able to exploit the information from the environment. To balance these two, the epsilon (ϵ) greedy strategy is introduced. This strategy ensures that the agent starts learning once enough information is gathered. It is implemented with the help of an exploration rate, where ϵ refers to the probability that the agent will explore [58].

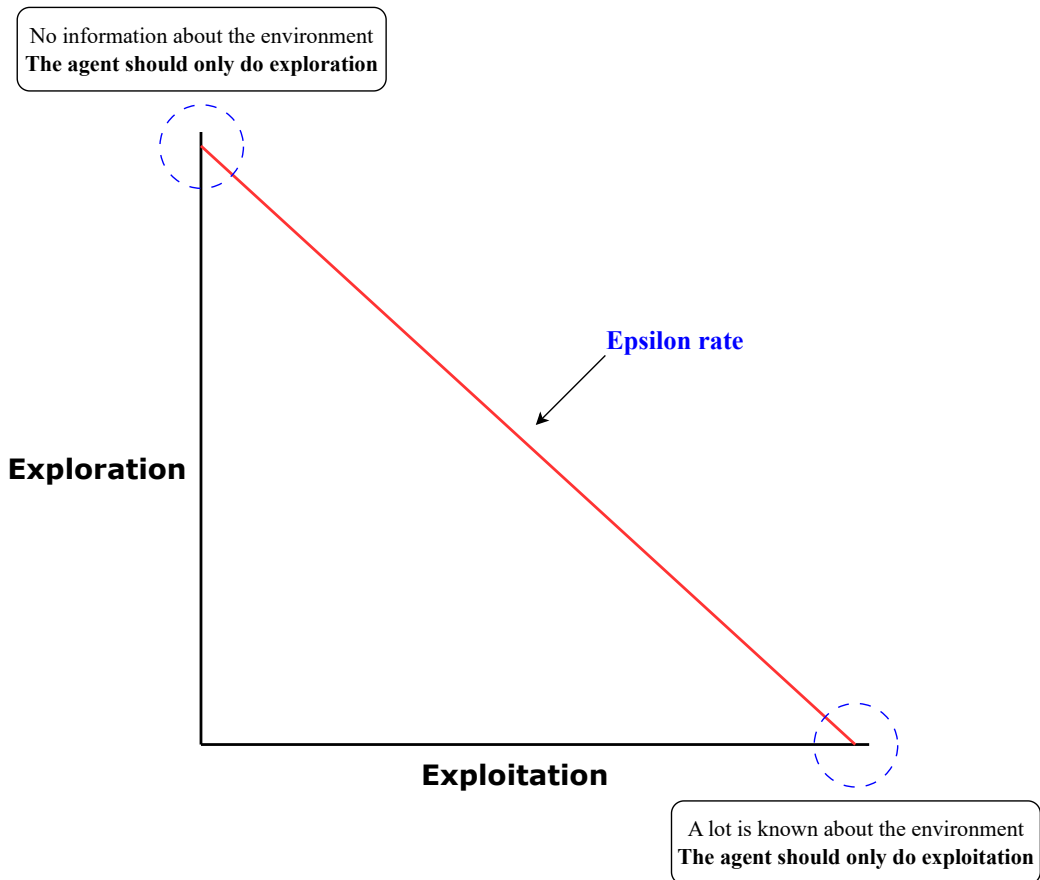


Figure A.3: The graph illustrates the epsilon rate which decays over time as the agent gets to know the environment more

As seen in Fig. A.3, epsilon decays over time. At the start of the training ϵ is initialized as 1.0, as it is desired to make the agent explore as much as possible, due to the environment being unknown. At each time step a random number between 0 and 1 is generated, if the number is smaller than ϵ the agent will explore. If the random number generated is larger than ϵ the agent will base its next action on exploitation. As epsilon decays over time, the agent is considered to be more "greedy" by mostly exploiting, and there is enough information about the environment [58].

A.4 Behavior Tree Implementations

Algorithm 2: Implementation of the Tick function for a fallback node

Data: globalLink

```

1 async Task TickAsync(globalLink):
2   State ← NodeState.Running;
3   await Children[CurrentChildIndex].TickAsync(globalLink);
4   switch Children[CurrentChildIndex].State do
5     case NodeState.Success do
6       CurrentChildIndex ← 0;
7       State ← NodeState.Success; return;
8     end
9     case NodeState.Running do
10      State ← NodeState.Running; return;
11    end
12    case NodeState.Failure do
13      CurrentChildIndex++; break;
14    end
15  end
16  if CurrentChildIndex ≥ Children.Count then
17    CurrentChildIndex ← 0;
18    State ← NodeState.Failure;
19  end

```

A.5 XPlanar Library

To control the XPlanar motors, a client was implemented enabling the communication with the ADS protocol. This client allows sending and retrieving data from Beckhoff PLCs, which are connected to the XPlanar. The client requires an ADS address and port to connect to the server. It gives access to all of the variables in the PLC program used to control the XPlanar shuttles [59]. To communicate and program the XPlanars, a PLC library is designed that contains a message buffer. This message buffer gets updated with the desired command giving access to control the shuttles to move.

The ADS client updates the message buffer in the PLC program when an XPlanar command is sent. To determine which action to perform, a protocol has been designed to communicate with the PLC. This protocol describes the message structure between the ADS client and the PLC and ensures that the messages are decoded correctly in the PLC.

Header 1	Header 2	Header 3	Reserved	Num of commands	Commands
0xFF	0xFF	0x42	0x00	Num of commands	Commands array

Table A.2: PLC message structure.

Table A.2 shows how a message structure is defined allowing the user to send multiple

commands to perform in the same PLC cycle. The commands have a predefined length to encode and decode messages. Therefore, the length should match the message sent to the PLC to ensure the correct decoding of the messages.

Different types of commands are supported such as enable, disable, and reset planar. Additionally, move commands are supported requiring a shuttle index, target position, velocity, and acceleration.

Furthermore, the PLC library exposes different variables which store the position, velocity, status, and feedback of the shuttles. The ADS client uses those variables to read and update shuttle properties. The ADS protocol makes it possible to subscribe to PLC variable changes and call a callback function whenever the values get updated.

A.6 Homogeneous Transformation Matrix

The homogeneous transformation matrix representing rotations and translation is given by the following equation:

$$\mathbf{T} = \begin{bmatrix} R & \mathbf{d} \\ \mathbf{0} & 1 \end{bmatrix} \quad (\text{A.1})$$

where R is the rotation matrix and \mathbf{d} is the translation vector. In this case \mathbf{d} represents the position of the shuttle in x and y along with a desired z component as shown in the following equation:

$$\mathbf{d} = \begin{bmatrix} d_x & d_y & d_z \end{bmatrix}^T \quad (\text{A.2})$$

The rotation matrix R can be represented as a composition of rotations about x, y, z axes. The x -axis, y -axis and z -axis rotations are represented by the angle θ , ϕ , and ψ , respectively. The individual rotation matrices are defined as follows:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \quad (\text{A.3})$$

$$R_y(\phi) = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \quad (\text{A.4})$$

$$R_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (\text{A.5})$$

The rotation matrices are multiplied together to obtain the rotation matrix R :

$$R = R_z(\psi) \cdot R_y(\phi) \cdot R_x(\theta) \quad (\text{A.6})$$

After obtaining the homogeneous transformation matrix in the shuttle reference frame $T_{shuttleorigin}^{goal}$, it has to be transformed into the robot's reference frame. This is done by using the following equation:

$$T_{base}^{goal} = T_{base}^{world} \cdot T_{world}^{shuttleorigin} \cdot T_{shuttleorigin}^{goal} \quad (\text{A.7})$$

The base frame represents the robot's base frame and the shuttle origin frame represents the reference frame of the shuttles.

This matrix is given to RoboDK which will perform inverse kinematics on it to get the joint variables for the robot.