Tampere University

Teemu Mökkönen

AUTONOMOUS PALLET PICKING USING ROS2

Master of Science Thesis
Faculty of Engineering and Natural Sciences
Examiners: Reza Ghabcheloo
Jukka Yrjänäinen
January 2024

ABSTRACT

Teemu Mökkönen: Autonomous pallet picking using ROS2
Master of Science Thesis
Tampere University
Degree Programme in Automation Engineering, MSc.
January 2024

In industrial use cases, heavy-duty mobile machines are commonly used for agriculture and earth-moving. Automating these machines can be a complex task that can introduce many different architectural problems given the use case, the level of automation, and the environment. Many navigation and control paradigms can be followed when designing autonomous units, such as reactive, deliberative, behavior-based, and hybrid.

This thesis explores implementing a hybrid control architecture in the use case of pallet picking. The aim is to recognize the different components of the navigation system and divide them into a hybrid navigation system in the form of a layered architecture that can employ a decision-making layer for generalized mission deployment. Decision-making could be used with many tools, such as petri-nets and finite state machines. One such thing that has emerged more recently, called behavior trees, has been deployed in recent years from the gaming industry to try to increase the re-usability of the developed system components by utilizing minimal transition rules and states between the nodes in the tree structure.

The thesis aims to deploy the system using ROS2 as a middleware solution to distribute feedback and commands through the ROS2 platform application. The aim is also to recognize the algorithm packages and frameworks from the large ecosystem of different solutions in the ROS2 that can make deploying autonomous heavy-duty mobile machine systems faster and easier.

In conjunction with the layered architecture design, ROS2, and behavior trees, it is possible to recognize the machine primitives and actions and bind them to functionalities of the machine so that employing more complex task deployment such as pallet picking is possible with simple behaviors in the behavior tree, with the layered architecture. The machine controllers expose interfaces that the behavior tree nodes can utilize to fulfill the primitives of the action.

Finally, in the thesis, there is an evaluation of the performance of critical components for successful pallet picking in the realized system architecture. Since the application mainly depends on the performance of the path following, localization, state estimation, and manipulator trajectory tracking, they are under evaluation. In the end, there were successful attempts at the pallet-picking system, given the architecture and system deployment in the distributed control system in the target machine. However, the RCLPY implementation showed performance bottlenecks and poor scalability in the CPU performance, given the higher rate topics in the system.

Keywords: ROS2, pallet-picking, Behavior trees, navigation architectures, heavy-duty mobile machines

TIIVISTELMÄ

Teemu Mökkönen: Autonomous pallet picking using ROS2
Tampereen yliopisto
Automaatiotekniikan DI-ohjelma
Tammikuu 2024

Teollisissa käyttötarkoituksissa raskaita liikkuvia koneita käytetään yleisesti metsätaloudessa, maataloudessa ja maansiirtotöissä. Näiden koneiden automatisointi voi olla monimutkainen tehtävä, joka voi tuoda mukanaan monia erilaisia arkkitehtonisia ongelmia käyttötapahtuman, automaation tason ja ympäristön mukaan. Autonomaattisten yksiköiden suunnittelussa voidaan noudattaa monia erilaisia navigointi- ja ohjausparadigmoja, kuten reaktiivinen, harkittu, käyttäytymisperustainen ja hybridimalli.

Tämä opinnäytetyö tutkii hybridiohjausarkkitehtuurin toteutusta palkkien poimintakäyttötapauksessa. Tavoitteena on tunnistaa navigointijärjestelmän eri komponentit ja jakaa ne hybridinavigointijärjestelmään kerrosarkkitehtuurin muodossa, joka voi hyödyntää päätöksenteon kerrosta yleistetyn tehtävän suorittamiseksi. Päätöksenteko voidaan toteuttaa monilla erilaisilla työkaluilla, kuten Petri-verkoilla ja äärellisillä tilakoneilla. Yksi viime aikoina esiin noussut, niin kutsuttu käyttäytymispuu, on peräisin peliteollisuudesta ja pyrkii lisäämään kehitettyjen järjestelmäkomponenttien uudelleenkäytettävyyttä hyödyntämällä minimaalisia siirtymäsääntöjä ja tiloja puurakenteen solmujen välillä.

Opinnäytetyössä tavoitteena on käyttää ROS2:ta väliohjelmistona palautteen ja komentojen jakamiseen ROS2-alustasovelluksen kautta. Tavoitteena on myös tunnistaa algoritmi-paketit ja kehykset ROS2:n laajasta ratkaisuekosysteemista, jotka voivat nopeuttaa ja helpottaa autonomisten raskaiden liikkuvien koneiden järjestelmien käyttöönottoa.

Kerrosarkkitehtuurin suunnittelun, ROS2:n ja käyttäytymispuiden yhdistelmällä voidaan tunnistaa koneen primitiivit ja toiminnot ja kytkeä ne koneen toimintoihin, jotta monimutkaisempien tehtävien, kuten kuormalavojen poiminnan, toteuttaminen on mahdollista yksinkertaisilla käyttäytymisillä käyttäytymispuussa kerrosarkkitehtuurin avulla. Koneen ohjaimet tarjoavat rajapintoja, joita käyttäytymispuiden solmut voivat käyttää täyttääkseen toiminnon primitiivit.

Viimeiseksi opinnäytetyössä suoritetaan arviointi avainkomponenttien suorituskyvystä onnistuneen palkkien poiminnan saavuttamiseksi toteutetussa järjestelmäarkkitehtuurissa. Koska sovellus on suurelta osin riippuvainen polun seuraamisen, paikannuksen, tila-arvion ja manipulaattorin trajektorian seurannan suorituskyvystä, ne ovat arvioinnin kohteena. Lopulta palkkien poimintajärjestelmässä saavutettiin menestyksekkäitä tuloksia ottaen huomioon arkkitehtuuri ja järjestelmän käyttöönotto todellisessa kohdekoneessa hajautetussa ohjausjärjestelmässä, vaikka RCLPY-toteutus osoitti suorituskykyongelmia ja heikkoa skaalautuvuutta CPU-suorituskyvylle ottaen huomioon järjestelmän korkean taajuuden kommunikaatioaiheet.

Avainsanat: ROS2, kuormalavojen poiminta, behavior trees, navigaatio arkkitehtuurit, raskaat työkoneet

PREFACE

I would like to express my gratitude to Reza Ghabcheloo and Jukka Yrjänäinen for offering me a chance to do my master's thesis on an interesting and challenging topic under their guidance and supervision.

Additionally, I would like to thank everyone who has worked with me on related topics and projects to achieve the current state of the work on the autonomous machine. Finally, I would also like to thank everyone with whom I worked and met during my studies; many of the goals were achieved together with you all.

This thesis has been done under the PEAMS (Platform Economy of Autonomous Mobile Machines Software Development) in the Autonomous Mobile Machine Group at Tampere University.

, 29th January 2024

Teemu Mökkönen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
BT	Behavior Tree
CAN	Control Area Network
CARMEN	Carnegie Mellon Robot Navigation Toolkit
CPU	Central Processing Unit
DDS	Data Distributed services
DEDS	Discrete Event Dynamic System
EKF	Extended Kalman Filter
FCL	Fast Collision Library
FSM	Finite state machine
GNSS	Global Navigation Satellite system
HDMM	Heavy-duty mobile machine
HFSM	Hierarchical Finite state machine
IDL	Interactive Data Language
IMU	Inertial Measurement Unit
KDL	Kinematics and Dynamics Library
LCM	Lightweight Communication and Marshalling
LIDAR	Light Detection and Ranging
LoA	Levels of Automation
OMPL	The Open Motion Planning Library
RCL	ROS Client Library
RCLCPP	ROS Client Library for C++
RCLPY	ROS Client Library for Python
REP	ROS enhancement proposals
RMW	ROS Middleware
ROS	Robot operating system
ROS2	Robot operating system 2

RTK	Real Time Kinematic
Rviz2	ROS Visualization 2
SLAM	Simultaneous Localization And Mapping
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UKF	Unscented Kalman Filter
XML	Extensible Markup Language
YARP	Yet Another Robotics Platform

1. INTRODUCTION

Heavy-duty mobile machines (HDMMs) can be used for multiple applications, from storage houses to farming. Automating these machines would allow repetitive tasks to be performed without human intervention. Different use cases vary according to the environment and the machine, but many places do or could utilize automation. Not all machines need to be fully automated, but even operator assistance in some operations can be considered automation [1].

Autonomous mobile robots and heavy-duty machines are widely used systems capable of planning their way between multiple goal points and tasks in their designed environments. These systems usually need to be able to define answers to the following questions before being systems that can perform actions autonomously: "Where am I?", "Where do I go?", and "How do I go there?" [2]. This means that the system needs to be aware of its location, surroundings, goal position in the designed environment, and how it will reach that position. In general, the question of how to reach the goal also means getting to the goal without colliding with the surrounding environment. This leads to a more complicated definition of navigation and control architecture division in the system in the partly observed environment. There will be erroneous situations in the partly observed environments, like blocked generated paths. These situations are nearly impossible to avoid in dynamic environments. In addition to the system being able to avoid these situations, it should be able to recover from them by itself.

Autonomous units with an increasing number of components also bring complexity into the design of the systems. This complexity comes from algorithms and components that need to be combined to enable multiple actions, operations, and error handling of the autonomous system. Coordinating the autonomous actions of navigation and manipulation while modeling the environment and perceiving the key points in the environment brings complexity to the system, which needs to be solved and coordinated. Additionally, complexity can introduce many different challenges to the implementation of the end product for the autonomous units that can be hard to handle or extend later on. Smart architecture design and implementation can help sort the system's complexity into logical components in a structured manner so that it is easier to maintain and extend later on during the system's development lifecycle. In the autonomous system architecture, one of the design points would also be considering decision-making given the high-level appli-

cation, use case, and coordination of the system to low-level actions. The system should be extendable to execute many different tasks than one single task and be robust and reliable so that system faults can be resolved autonomously.

Many software architectures or frameworks have emerged offering solutions to simplify autonomous system development and decrease programming repetition by offering general abstract interfaces, middleware solutions, and algorithms. One of these frameworks is ROS2 (Robot Operating System 2), which offers accessible real-time communication for robot systems [3]. ROS2 and its utilities can be combined to make a navigation architecture to plan its route to the objective point and complete pre-defined tasks such as pallet picking. ROS2 offers collections of algorithms and frameworks for state estimation, navigation and control like TF2, navigation2, moveIt2, and robot localization, which were in key roles while developing the pallet-picking use case [3][4][5][6][7].

Research questions were set to support the research of autonomous system architecture and decision-making design while evaluating ROS2 as a middleware solution. The research questions are as follows.

- What kind of control paradigm could be used to implement a high automation level application for pallet picking, given the initial uncertain estimation of pallet location?
- How could one such paradigm make decision-making in the context of pallet picking with ROS2?
- What implementations of frameworks and algorithms does ROS2 offer that support achieving autonomous pallet-picking tasks easier or faster?
- How does the implemented architecture perform on the given task, and what are the key points for a successful pallet picking and autonomous architecture?

The research methodology used is the design science methodology, where the problem to be solved is how to design the pallet picking architecture [8]. The objective of the architecture is to have a modular ROS2 architecture that meets the requirements set for the application. The realized system is then evaluated in contrast to the architecture design properties and experiments that measure systems capabilities in the given task and the system load overhead with the tools used in the system.

This thesis guides the reader through the general navigation and control architecture and how it can be designed using ROS2 as an intermediate communication platform or middleware. This current chapter (1) provides the reader with a general overview of the thesis, research questions, and relevant topics. The second chapter (2) introduces the reader to what an automated heavy-duty mobile machine is, what the levels of automation in HDMMs are, and what kind of system architecture components are required for a high level of automation. Chapter three (3) goes through layered architecture that can be utilized to make one navigation and control paradigm that recognizes the decision layer

as one of the layers to perform autonomous navigation with decision-making and system coordination. This chapter aims to provide background to the second research question from the point of view of the control architecture while introducing the tools that could be used for decision-making. Chapter four (4) introduces the ROS2 middleware component for the architecture and the tools used in the work that the ROS2 ecosystem provides for developers, which answers the third research question. Chapter five (5) contains the details for the work done with heavy-duty wheel mobile machine pallet picking, with the done behavior tree, while introducing the details, features, and requirements for the pallet-picking application. The sixth (6) chapter introduces the experiments, environment, and test results done in the thesis. This provides an answer to evaluating the performance of the implemented architecture. The seventh (7) chapter evaluates the realized pallet-picking architecture given the architecture design properties or attributes learned through the experiences and experiments with the architecture. Chapter eight (8) is the conclusion for the work, the presented ideas, and any future works that could be done based on this work.

2. AUTONOMOUS HEAVY-DUTY MOBILE MACHINERY

The purpose of this chapter is to provide an introduction to heavy-duty mobile machines the architectural design required for machines to be automated, and what are the automation levels. Capabilities of self-driving machines are often represented in levels of automation [9]. The aim is to introduce automating these machines as a problem that requires careful planning, even at the architectural level. This chapter provides some background on autonomous heavy-duty mobile machine (HDMM) systems relevant to the thesis while giving motivation to the other topics discussed in the thesis.

Heavy-duty mobile machines have been used for a long time in industrial use cases such as foresting, earth moving, and farming [1]. In figure 2.1, the common heavy-duty mobile machine systems used in the industry can be seen for different purposes.

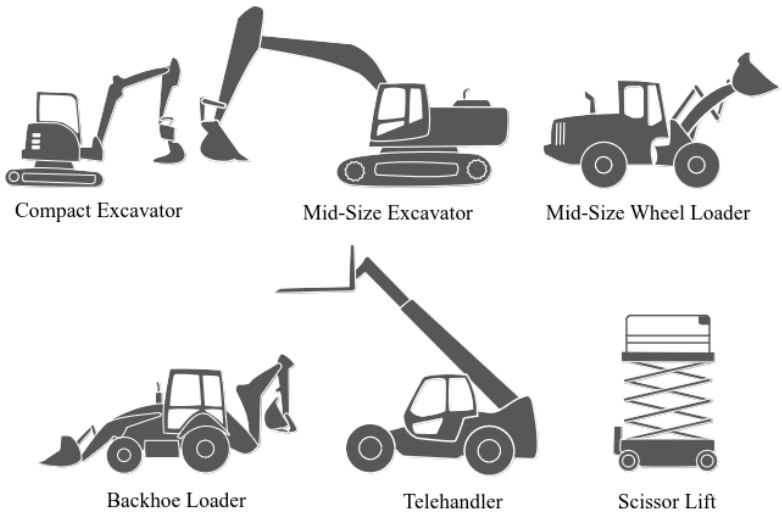

Figure 2.1. *Common HDMMs types used in the industry [10]*

These machines require experienced operators [1]. The need for experienced operators can be mitigated with automation in the HDMM system [11], which can help master the machine's use faster. Automation and autonomy can be considered at multiple different levels of automation (LoA) [9] [1]. Automation can already be experienced in many machines in many other parts and systems. HDMMs implement two high-level tasks that can be automated: navigation and manipulation. Implementing operator assistance can be a simple task to bring a low level of automation into the system. Automation can affect

only one part of the system, which can be seen, for example, in the cruise control of the machine.

High automation level applications can execute numerous different tasks without human intervention. Managing such a system requires numerous different components so that all the system algorithms and tools work together to achieve a common goal. This combination of various components can be complicated to manage and implement. To accomplish this common task, there comes a need to understand the needs for the HDMMs that were introduced in the chapter 1 as three sentences that form a basis for many robotics applications: "Where am I?", "Where should I be?", and "How do I get there?". At the same time, these sentences describe the needs for many robotics applications required for autonomous operations. Reaching a higher level of automation requires considering the system architecture, where the system's needs could be defined based on the target use case requirements and the capabilities of the robot platform itself.

2.1 Levels of automation

There are standards and legislation to define a taxonomy for autonomous vehicles that aim to divide the levels of automation. In SAE J3016, the levels of automation have been divided into six levels ranging from zero automation to fully autonomous unit [12]. The scope of this standard includes motor driving automation systems that partly or entirely perform driving actions in the system [12]. As previously stated, the HDMMs consist of two autonomous tasks: navigation and manipulation. This standard defines the taxonomy only for navigation tasks.

To describe the levels of automation more accurately for autonomous HDMMs, there have been efforts to standardize the taxonomy [1]. Machado et al. (2021) divide the levels of automation into a 6x6 matrix, as can be seen from figure 2.2, where one axis represents the machine's autonomous driving and other manipulation capabilities.

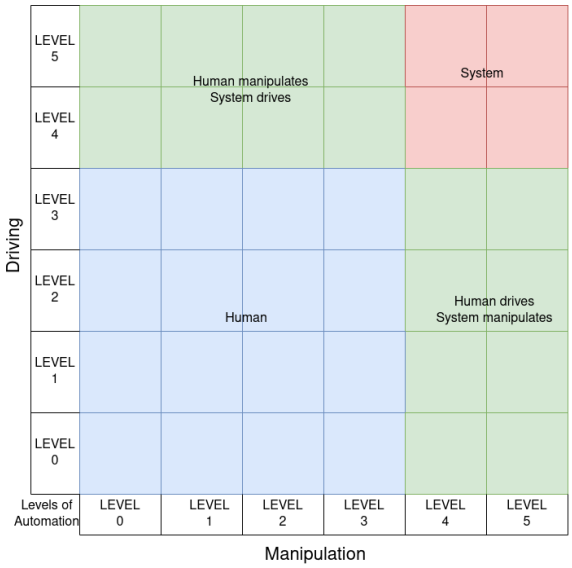

Figure 2.2. *Levels of automation matrix [1]*

In the matrix, the levels go as follows in the table 2.1, which is adapted from Machado et al. (2021)[1]:

Level	Navigation	Manipulation
0	The operator is responsible for perception, planning, and control.	The operator controls all manipulator joints separately.
1	One driving assistance functionality at a time.	Human is controlling end-effector in Cartesian space.
2	The machine can control at least two different driving functionalities the operator assigns.	Machine implements simple trajectory following predefined trajectories that require activation sequence.
3	Can execute the specified task with a predefined path while being able to execute obstacle avoidance and emergency stop while perceiving environment.	Can perform tasks on its own if the conditions are met. It needs authorization from the operator to execute actions while perceiving the environment.
4	Able to perform decision-making and SLAM (Simultaneous Localization and Mapping) in structured environments. Can recover from e-stop or avoid obstacles dynamically. The machine performs high-level tasks with a high perception of the environment and decision-making. The system can gather and generate world models from the sensor data.	Extension on level 3, while being able to plan and execute collision-free, while being able to request human intervention if the conditions for the execution are poor. The machine executes high-level tasks with a high perception of the environment and decision-making. The system can gather and generate world models from the sensor data.
5	Active localization and mapping. The system can execute high-level decision-making like path generation and obstacle avoidance. It can operate in space with other autonomous units and all weather conditions. Ability to operate under any conditions autonomously.	Able to perform in any conditions while communicating and coordinating with other machines around it and operating with external materials. Able to operate under any conditions autonomously.

Table 2.1. *Levels of automation in navigation and manipulation[1]*

As seen from the list, the lower levels of automation implement simple operator assistance applications. Still, when going to the higher levels like from 4 and 5, the machine is required to have higher-level decision-making and task deconstruction to be able to perform in the given high-level missions in addition to simple navigation and manipulation tasks with additional reliable and robust scenario handling, where the system can recover

from edge cases under which it could be operating in. This higher-level automation brings more system components and applications, which, in turn, need to be built, maintained, and extended, increasing the system's complexity [13].

2.2 Autonomous HDMM architecture

Autonomous HDMM architecture requires integrating multiple system components to operate autonomously without the human operator's external aid. The taxonomy of the components differs from one source to another from the division of the components into three [14][15] to five [16] different parts. The division of three, where system components are divided into perception, planning, and control, fits the fundamental questions mentioned previously [15][14].

"Where am I?" is a metaphor for the perception part of the system, where the system needs to be able to perceive where it is and the surrounding environment, which can include environmental modeling based on the sensor data. Representation of the environmental modeling ranges from high-level symbolic representation to the more accurate semantic categorization of the data to understand the world more accurately by defining obstacles, roads, and traffic signs, to name a few [15]. This categorization can also include modeling the manipulation target in the HDMMs workspace.

"Where should I be?" and "How do I get there?" together form a metaphor for the planning and control parts where the planning consists of mission level plan to the motion plan level to achieve the higher primitives in the robotics applications and route them to compound actions of the system [15][17]. In this context, the primitives represent high-level task definitions such as a "Pick a pallet," and compounds are less abstracted actions or instructions that still need to be refined to actual commands to the machine platform, such as a path to be followed. A path does not tell the platform how fast it should be driven or steered, but it gives the guidelines for how the machine should act. Hence, it offers an instruction set that should be refined to actual commands to the platform. Control refers to forming the compounds to commands in the robotics platform that can be executed by the platform into actions of the machine. Actions are physical phenomena executed by the machine to the environment. Combining these components can develop architectural problems in the navigation and control system where there are many subproblems on their own. Combination can appear in the figure 2.3.

In the figure, the perception component gathers relevant world information that could include world model and vehicle pose to name a few. Perceived information is then fed to the machine's planning interface. The planning interface collects the goal, state, and world representation. Compounds are then formed and sent to the control, which uses the compounds to form commands for the robot platform. The platform turns the commands to physical actions in the environment [17][15].

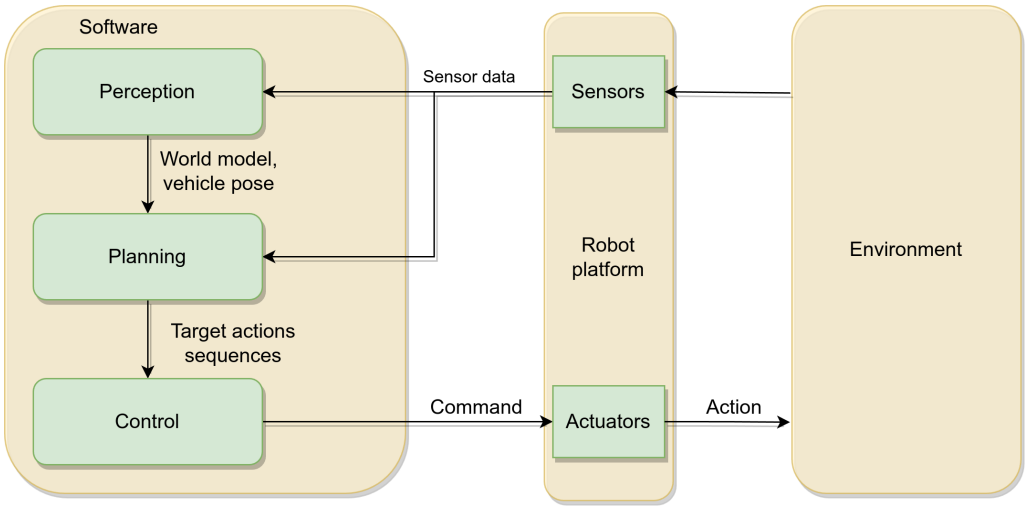

Figure 2.3. *Combining the different parts to autonomous application*

The components in an autonomous machine must be fitted with each other so that the system can utilize the gathered data to form concise actions to accomplish the given task. Two of the major roles of architectural design in many applications are to increase the flexibility and adaption in the given task [18]. Adaptation of the robot can be implemented in multiple levels of the robot architecture. In high-level planning, adaptation can mean changing the mission definition according to the changes in the dynamic world or the machine's state. Adaptation can be done in the lower level part of the control system where the existing compounds constitute the action-decision methodology [18].

There are many design properties to consider when designing the control and navigation architecture to be considered: *reliability, generalization, modularity, autonomy, extensibility, reactivity, and run-time flexibility* [13][19][20], which also overlap highly with the software architectural design properties. *Modularity* refers to the possible interchangeable components of the system, whereas in a highly modular system, components should be interchangeable with different components. *Extensibility* refers to the ability of the system to be extended and the effort to perform the extension to the system or machine. *Autonomy* in the design refers to how well the system should perform alone in the given tasks without human intervention. *Generalization, run-time flexibility, robustness, and reactivity* together are designed according to the needs of the system to react with imperfect inputs to the system, such as broken or faulty sensor data, respond to different states of the system in known and unknown situations, and to the performance of the system to reconfigure the during the task execution. When making the intelligent design of the system architecture, these questions can be answered, and the system's complexity and management can be easier to handle and extend. These design properties are often considered through the needs of the robot platform and application requirements, where specific system compositions have certain advantages over others in the robotics designs [20].

3. LAYERED ARCHITECTURE AND TASK PLANNING

Large robotics application tasks like mobile manipulation consist of multiple tasks that robots must execute in parallel, such as perception, planning, control, and monitoring. There are general control navigation architectures and paradigms for robotics applications such as deliberative, reactive, behavior-based, and hybrid [13][21][2]. The hybrid control architecture is a common one that implements high automation level applications that generally follow the formulation in the previous chapter 2.2[16]. Figure 3.1 shows the graph behind the hybrid control system.

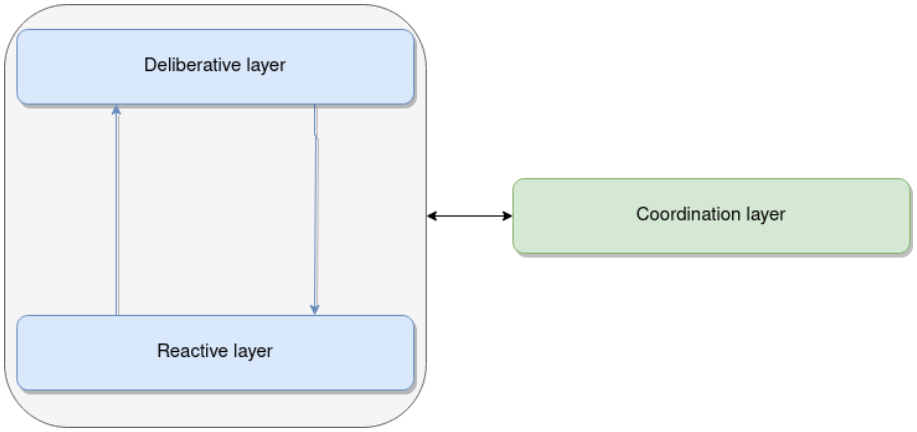

Figure 3.1. *Hybrid control architecture example*

Hybrid architectures aim to reduce the problems with reactive and deliberative layers in the system and create architecture that can operate under incomplete data and react fast to changes in the environment [22, p. 43][23][16, p. 309]. Since the deliberative and reactive layers most often operate under different time constraints, the decision-hybrid layer coordinates the layers to function together to provide the cohesive output of the system [16, p. 309][2, p. 11].

Designing a robotics software architecture comes with a need to divide the system into logical parts that can be separated. This is where some have adopted the layered architecture, which divides the system control logic into three different layers as the name suggests [24][25]. The three layers are named **decision**, **executive**, and **functional** layers. Layered architecture is a common way to design and implement hybrid control [24][25]. This layered architecture employs the coordination layer to coordinate the hybrid

architecture's functional and executive layers, which operate similarly to the deliberative and reactive architectures [13]. This coordination layer can also be utilized for mission-level task planning, where the layer's components can be coordinated according to the needs of the goal or mission. Other intermediate layers are used in large robotics applications like the middleware and hardware layer [26], which can encapsulate high-level architectural components, such as system components in layered architecture.

Task planning is an act of having a high-level goal or primitive for the system that needs to be divided into lower-level actions or compounds that the system can handle [16, p. 284]. These low-level tasks can be, in some cases, considered as skills for the robot where the skill is represented in a symbolic level and motion level [27][24][25]. Tasks like moving pallets from pose A to pose B are easy for humans to understand and divide into logical small actions. This is where the task planning comes in for autonomous units to coordinate the system execution and decision-making.

The layered architecture allows for many design principles in the architecture of autonomous robotics systems, such as modularity, extensibility, reactivity, generalization, and even autonomy. Given the proper implementation and distribution of the modules in the system, the modules should be modular and extendable. The correct formulation of the functional and reactive layers enables the system to be general for different HDMMs, where the decision layer employs case-related functionalities and coordination. In contrast, the deliberation layer handles solely the world representation parsing to compound actions, while the reactive layer refines these compounds. Correctly formed architecture should end with a modular product that can perform the given tasks. Still, it should also be extendable to other applications and use cases without the need to refactor the whole base architecture.

This chapter looks at layered architectures and how they are designed in robotics applications. This includes defining the layers and their tasks in the architecture. After this, there will be a more detailed look into the common tools to generate the decision layer for the coordination layer of the architecture. Although the mission description can be divided into two subgroups, logical languages and discrete event dynamic system specification languages, this thesis will handle the dynamic system specification languages (DEDS) specifically [24]. The main DEDS specification languages that arose from literature for mission planning were behavior trees (BT), finite state machines (FSM), and Petri nets [24].

3.1 Functional layer

The functional layer is between the machine-specific hardware and the executive layer [28]. This layer creates interfaces between sensors and actuators and provides this data to all modules in the system according to the predefined interface provider. The func-

tional layer transfers the sensor data to a common interface format so other layers can use the information later. On the actuator level, the functional layer provides a way to turn higher-level instructions into commands like movement on the actuator level [16, p. 289][24]. The functional layer can be quite large and contain multiple submodules or controllers, as shown in figure 3.2, which shows one reactive architecture called subsumption architecture. The functional layer provides an interface from something high-level, like a path following the hardware level, like actuators, which can have multiple controller modules in between in this one layer. Since the functional layer acts as a bridge between the common system interface and the control system, this layer can be very machine-specific and hard to generalize to multiple machines. Unless the sensors and low-level actuators are very similar, and in many cases, the abstraction of the interfaces is more challenging to maintain the lower in the architecture we are. This means that the lower the system the algorithm is in the system architecture, the harder it is to be interchangeable between robotics platforms due to the platform's needs. Traditionally, the functional layer can take high-level commands such as acceleration or velocity input of the machine in the body frame and convert them to actuator-level commands such as torque or voltage.

Subsumption architecture can be described as layered control where each layer level can override the previous layer according to information from the environment that the robot receives. Subsumption architecture could implement part of the functional layer. Figure 3.2 shows how subsumption is layered where the lower levels can override higher ones.

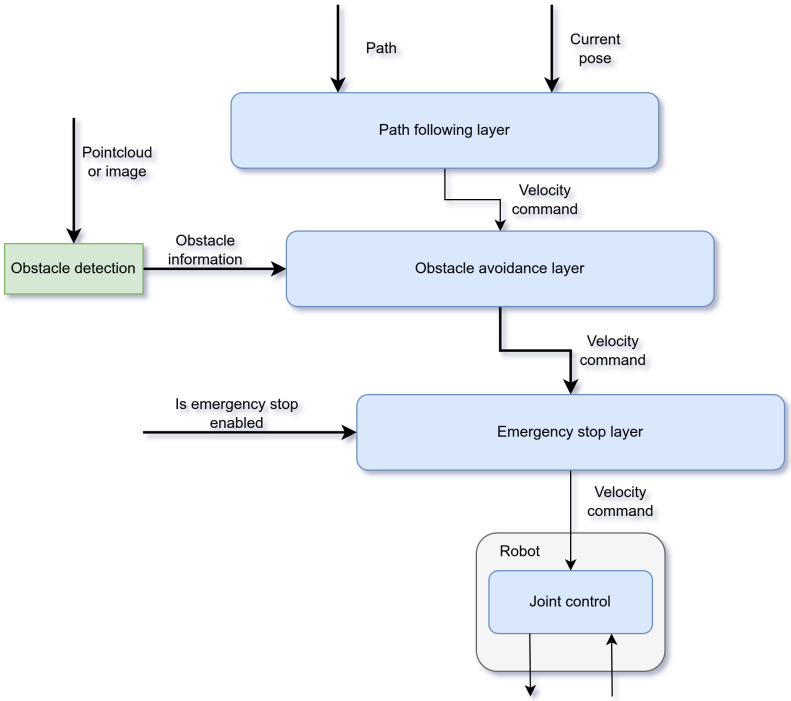

Figure 3.2. *Subsumption example architecture for path following control*

In a path following control scenario, the same figure 3.2 can be a three-layer subsumption

architecture. The first layer could be a simple path following the control layer, and the second could be an obstacle avoidance layer, as seen in figure 3.2. In this case, the path following controller would pass some control input to the mobile machine. At any point, the obstacle avoidance could override the command according to the sensor data and make its safe command to follow the path without collision [22, p. 93]. Then, the last layer, the software-level emergency stop, could be an extra component that could read human input to the system and see if an emergency stop would be enabled. However, this component would not usually be implemented at the software level. This means that obstacle avoidance would be prioritized over path-following commands during the operation, and the emergency stop would have the highest priority. There could also be more layers that have different purposes in the navigation. In the example, each layer can run at a different rate during the navigation, and usually, the control frequency increases closer to the robot platform we go.

In the context of the control architectures, the functional layer would implement the reactive layer, a fast layer with non-blocking actions and access to the sensor data layer that provides immediate feedback [25]. The reactive layer could implement a subsumption architecture that would follow the path following commands while monitoring and avoiding obstacles. This layer is also responsible for monitoring the feasibility of the given task. This layer must inform if the assigned task is not feasible to achieve from the current state. This information needs to be readable from the decision layer to resolve the issue [16, p. 289]. This means that during the control sequence in a functional layer, in the event of the machine deviating too far from the given instruction set or path or the given path being blocked, the functional layer would be able to signal this to the decision layer.

The functional layer should implement simple high-level command interfaces that accept the compounds in the form of, for example, paths or trajectories for navigation and manipulation separately or together, depending on the architecture. This increases the extensibility to other tasks when the high-level components instruct the functional layer to implement specific compound actions. There should be two-way communication where the functional layer provides feedback to the higher-level layers to offer more robust behavior in case of erroneous situations.

3.2 Executive layer

The executive layer takes high-level commands from the decision layer and, in turn, generates abstract solutions to these tasks. The executive layer is also responsible for converting the gathered world information to compound instructions set as paths, for example, to the lower-level components. This higher-level plan with the world model is often reusable in the many use cases, given that the system dynamics and kinematics are relatively the same. This means that the executive layer is responsible for checking that realizing the

task goal is possible from the current state [24]. In an actual application or software stack, the part of the executive layer could be a path planner for a mobile robot that generates a path or trajectory from the current location to the goal state. Suppose it is impossible to create a path according to the gathered world information like a map. This will be informed to the higher decision layer, which can create additional recovery tasks. These tasks could include informing the remote operator that the mission cannot be executed due to an external issue.

This layer acts as a deliberative layer in the control architecture and can have lower time constraints compared to the functional layer [25]. In an autonomous system, this layer can dynamically create new paths or trajectories during runtime asynchronously or optionally generate a new trajectory when the current path or plan cannot be executed fully or is required in case of a unique external event. The executive layer is also responsible for parsing world models to actual compound actions for the system to execute according to the perception system. For example, perception data could be represented as a grid-based map or voxel grid [29]. This layer should have access to the perception output of the sensing system while being able to understand it and utilize it for planning the next mission.

The deliberative layer is helpful in long-term navigation where the environment is complicated, and there could be multiple local minima where robots could get stuck by only reading the current sensor information from the environment as it would in the functional or reactive layer [22][30]. This deliberative layer in the system tries to answer with long-term plans, while it can be slow to respond to environmental changes. Deliberation, as the name suggests, also helps monitor the system since deliberative plans and actions can usually be visualized in the form of paths.

3.3 Decision layer

The decision layer is responsible for task planning and allocating the tasks to the correct components in the system. This layer provides the hybrid layer into the system, combining the reactive and deliberative layers. In a regular collaboration between a robot and a human, this layer would be handled by a human providing some commands to the robot with a user interface. Still, this layer can also be programmed to handle error states and provide a connection between different behaviors [24]. The secondary target for the decision layer is to maintain the system state so that the system can recover from the incorrect state in case of erroneous data or system error, increasing the system robustness and reliability [24]. In the case of mobile robot manipulation, this can mean, for example, replanting the path in case the original path is invalid or managing resetting the system state if it is not valid. As stated before, the decision layer is not responsible for keeping track of the erroneous situations in the system but only coordinates the recovery of the

system components if these events happen.

The decision layer could also be utilized to provide higher mission-level programming of the robot if the "skills" of the robot are generalized into the decision layer representation [24]. Given the abstract skills of the robot, such as "detect target item" and "move arm to target," the usage of the machine can be generalized, and the machine's skills are reusable in tasks other than the original primitive.

Generally, this means that the decision layer encodes the mission description instead of forming it; this means that the decision layer does not include the task or mission description and break it down to motion primitives, but it encodes the task primitives for the decision layer to perform and coordinate the mission. This means that mission deconstruction is still broken down in some form into the decision layer, which is coordinated to the lower-level layers according to the logical instruction set. Forming the mission or task definition is done by humans. It has also been implemented by reinforcement learning and evolutionary algorithms to adapt to specific tasks [24].

There are many ways to design this layer. Still, this section will look into a couple of those implemented in robotics applications like behavior trees and finite state machines and could be targeted to enable modular design with ROS2. Behavior trees and state machines have been used previously to implement the hybrid layer to the system in mobile manipulation tasks [24]. In the following two sections, we will present the properties of the behavior trees and finite state machines.

3.3.1 Behavior trees

Behavior trees are modular tools for creating policy or control over robots or other actors like NPC (Non-Playable Characters) in games. Behavior trees originate from computer games, but later on, they were adopted by robotics and AI due to high modularity and re-usability [31]. The idea behind behavior trees is to represent actors' behavioral models as a tree, where each branch is its action that the actor can take if the conditions of the actor and environment are correct [32]. Behavior trees offer high modularity since each branch and leaf are easily interchangeable depending on the application.

Generally, six different **node** types are used in behavior trees. In table 3.1 can be seen all general node types and their uses [32][31]. This table follows the same format as table 1.1. in [31].environment

Node	Succeeds	Fails	Running	Number of children
Sequence	One child node must succeed	All children fail	If one child is running	[1 - N]
Fallback	All children node succeed	One child fails	If one child is running	2
Parallel	If $\geq$ M children succeed	If > N - M children fail	Else	[1 - N]
Action	When action is successful	When action fails	While the action is not done	0
Condition	If true	If false	Never	0
decorator	Custom	Custom	Custom	Custom

Table 3.1. *Behavior tree nodes*

By defining a small amount of interfacing between nodes, you can increase the system's modularity by creating easy-to-combine small components. As can be seen in Table 3.1, there are virtually three different states that each node can return: *success, failure,* and *running*.

Applying these six simple nodes in behavior trees can take on complex assignments. The flow of the tree is controlled by *sequence*, *fallback*, and *parallel* nodes, which are also referred to as **control nodes** of the tree structure. Real-world actions or commands taken by the robot are handled by the **action nodes**, which commence command when specific criteria are met. **Condition nodes** are the ones that decide if actions are to be taken and can be set freely by the user. For example, the condition could be "Is_battery_empty" to check the battery status before giving instructions to start navigating. **Decorator nodes** are usually custom nodes that change the outcome of the tree in some way. For example, the decorator could tick the child node continuously at a specific rate. Figure 3.3 can be seen as an example of a tree representation of the behavior tree.

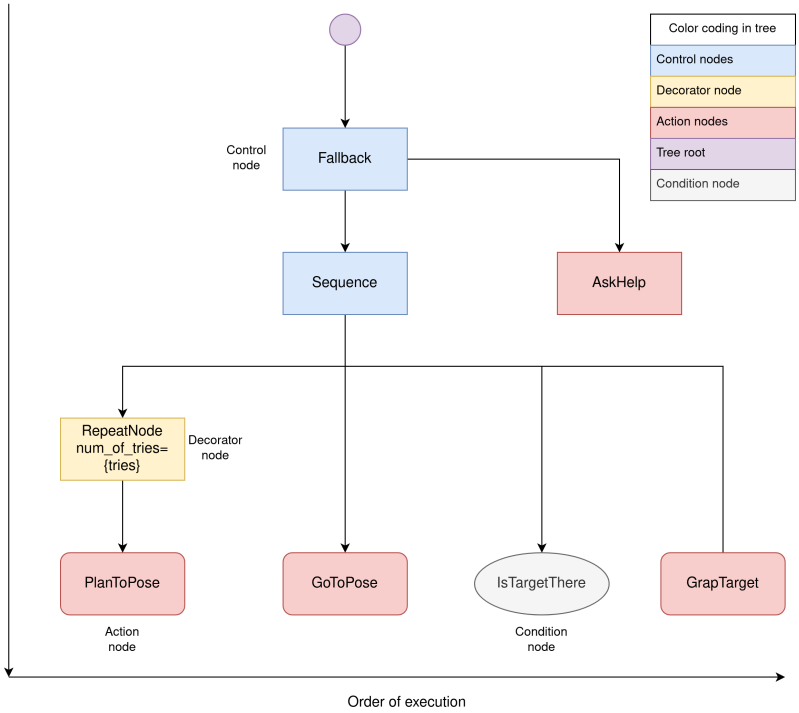

Figure 3.3. *Simple example of BT*

Figure 3.3 represents a simple example of behavior trees. This example consists of a navigation problem from pose to pose where the goal is to grab an object. This behavior combination has been simplified a fair amount and usually requires a lot of background processing from the other layers of the system. Tree execution begins from the tree's root, the purple circle on top. The first blue node in the graph after the root node is a fallback node, which will be called in case the sequence returns failure and executes the action "AskHelp", which would need to be getting human attention to the task. The second blue block is the sequence node, which will tick all its children from left to right if $N-1$ children return to state as success. If all went successfully, from left to right, the flow of the tree would go as follows: first, the decorator "RepeatNode" would be called, which will repeat planning until the path has been successfully generated or until the number of tries has been exceeded. Next would be initiated the "GoToPose", which tries to make the robot move to a planned pose. Next is condition node "IsTargetThere" which will try to check for the target and decide if the target is in the correct position. If the target is found, the robot or actor will grab it with the action "GrapTarget". From this simple example, it can be seen that it is pretty simple to generate behavior models for rather complex systems by using behavior trees.

As seen from 3.1, behavior trees can return *"success"*, *"failure"*, and *"running"*, which are three different states. Using simple return states, this method allows behavior trees to be modular [31]. However, an additional component is needed to handle information transmission between the tree nodes due to limited information flow in the tree structure.

This additional component can also keep track of the internal state representation of the machine at the cost of increased coupling between the behaviors in the tree structure. Behavior trees generally implement different "platforms" for dealing with information. This usually means something called blackboard. Blackboard implements a virtual key-entry storage that keeps track of external inputs and outputs to each tree node [32]. In conventional programming languages like Python or C++ and many others, blackboard could be described as a global dictionary that can be accessed anywhere from the tree structure. This means that each entry in the blackboard is a key-value pair, where the blackboard saves a value behind a particular key, which all of the behavior tree nodes have access to. Figure (3.4) is a simple example of how blackboard communication looks in the tree representation. Many of the behavior tree libraries offer this blackboard functionality. Still, most often, the blackboard cannot be localized between specific functions, and all entries are available to all nodes in the tree, which can cause issues with unintentional name clashing when the BT grows larger [33].

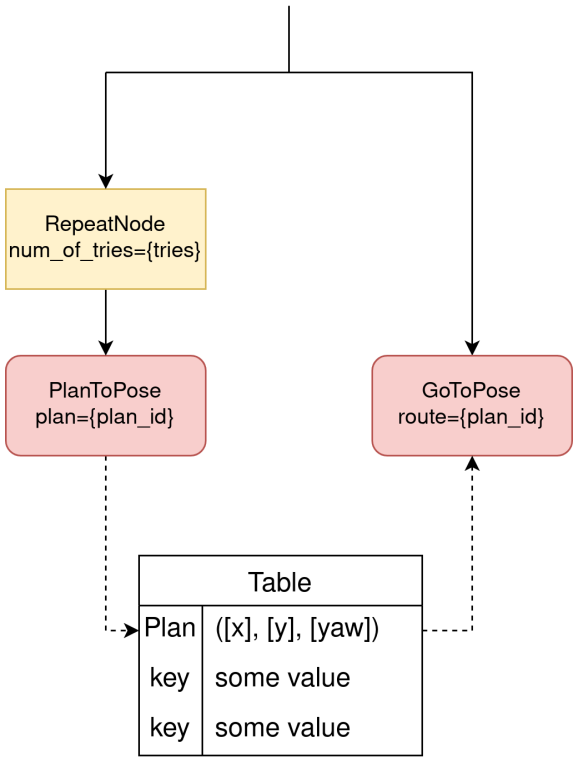

Figure 3.4. *Example of behavior node writing an entry behind key in blackboard dictionary that is accessed by another behavior node*

Figure 3.4 could be part of the simple behavior tree example 3.3 from the lower left corner. The communication between nodes can be seen here when it is needed. ’PlanToPose” will generate a plan that consists of a tuple containing a list of pose information ([x], [y], [yaw]) and save it behind the key value ”plan_id” to the blackboard. After this ”GoToPose” will read the value behind the key and start navigating according to the information on the blackboard.

3.3.2 Finite state machines

Finite state machines are mathematical representations of the states, originating from the gaming industry, that have transitions between them [31, p. 23][34, p. 2][35, p. 55]. Finite state machines have been applied to many different robotics applications, from manipulation tasks to bipedal robots[34, p. 2]. Finite state machines assume that the system could be in a limited number of states and that these states have a limited number of transitions between each other. State machines offer flexibility in designing robotics architectures by providing a way to maintain simple abstract states and create state transitions between states.

There are advantages and disadvantages to using FSMs. Simple FSMs are easy to implement and understand initially, and many FSMs use standard structure [31, p. 23]. Although these systems are easy to understand initially when the number of states in the system increases, so does the number of transitions. When the system needs to be reactive, there must be many states and transitions, which can hinder the modularity of these systems, due to the high coupling between the modules. When inspecting large FSM and there is a need to add a new state to the system, there is a large job to gather all states that can transit into the new state, which hinders the scalability and re-usability of the FSMs largely [31, p. 24].

Since decreasing modularity is common in large FSM, a hierarchical finite state machine (HFSM) answers the complexity issue by adding behaviors or states that can contain sub-states and transitions. This means that on a high level, HFSMs can have high-level states like ”follow path”, which can consist of multiple sub-states. This decreases the state transition between high-level and low-level states, which lessens the need to reprogram state transition if there is a need to increase the number of states as the system grows larger. Although the number of transitions is lower than before, there is still a need to keep track of multiple states and transitions, which can be cumbersome when the state machine graphs in the high or low-level increase [31, p. 25].

In the figure 3.5 can be seen the same task decomposition as for the behavior tree in the image 3.3 adopted from example [33].

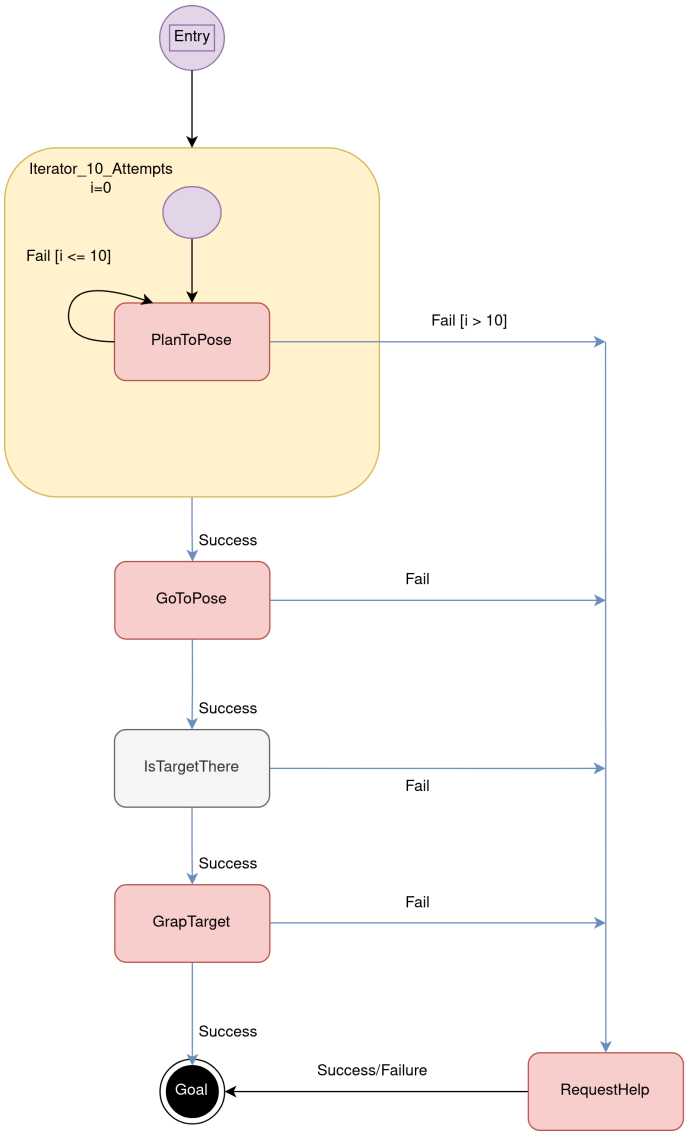

Figure 3.5. *Simple example of FSM*

The colors of each state and transition in the image represent the same rule set as in the behavior tree. For example, the Fallback transition is handled as a state in the representation where the system state transits, given the sequence pipeline's failure. The retry loop is set inside the composite state, which handles the retrying of the path generation, given the failure in the generation.

Some state machines adopt similar functionality to the blackboard called *UserData* that allows identical communication between the states in the system [33]. UserData implements a local input-output communication between the system states that have the key-value functionality [33]. This allows the communication to stay between the states, which avoids the global namespace clashing in small and large applications.

4. ROBOT OPERATING SYSTEM 2

Multiple software platforms have desired to create a modular robot design, programming, and communication framework. To name such a few, there were LCM (Lightweight Communications and Marshalling), YARP (Yet another Robotics Platform), and CARMEN (Carnegie Mellon Robot Navigation Toolkit) [3]. One of the newer and more active ones solving this problem is the Robot Operating System or ROS, which Willow Garage released in 2007 [3, p. 1]. ROS tries to decrease the repetition of programming general parts required for robotics applications, including communication and packaging of the system. ROS provides stable communication middleware-tailored protocol built on TCP/UDP (Transmission Control Protocol/User Datagram Protocol). There were some issues with ROS. For example, network topology and security were immature, so a new framework for robot programming called Robot Operating System 2 was built to be the successor of ROS [3]. ROS2 took a different approach by integrating Data Distributed Services (DDS), which aims to solve problems regarding network security, real-time operations, multi-robot communication, and communication in non-ideal network situations [3]. The first version of ROS2 Ardent Apalone was released in 2017 [36].

ROS2 is a framework for robot programming that is under an Apache 2.0 license, which makes ROS2 an open-source framework and can be broadly modified and distributed according to the projects' personal needs and distributed according to the needs of the project, without contributing to the ROS2 framework [3][37]. ROS2 can be divided into three different parts as a software ecosystem according to the [3]: middleware, developer tools, and algorithms.

The middleware part of ROS2 has been designed with security and reliability in mind [3]. This means that they have adopted DDS communication as their middleware, which brings more mature network topology, security, and quality of service (QoS) options to the developers [3]. What DDS also brings to the table is the ability to perform real-time communication [3], which can be considered as an advantage over ROS, which did not officially have the capabilities of real-time communication.

4.1 ROS2 communication and middleware concepts

ROS2 is divided into multiple concepts that can be used to create complex systems that can interface with robots, and sensors send communication messages over distributed systems and act as middleware for multiple components in the system. These concepts include, for example, nodes, topics, services, and actions. These concepts help to create modular and robust systems that can be used to develop easily replaceable components for each system.

4.1.1 Nodes and Topics

ROS2 handles all of its features as **nodes**. Usually, the node is an entity that should create one logical part of a complex system and communicate over the communication network with other nodes [38]. Together, all nodes form, for example, a complex control system for different types of robots. The number of nodes in each system depends on the implementation and the system's needs. As mentioned, each node can and should implement some logical part of the system; this can include, for example, reading sensor data and filtering it and sending it to other parts of the system, path following, path generation, or localization. The system developer needs to define each node themselves and divide them into logical parts. If this division is done correctly, there could, for example, be multiple nodes that each do the same task, but they could change in the manner of moments just by re-configuring new nodes to use the same interfaces. To abstract and make the communication between system components, ROS2 uses interface definition and language mapping (IDL) to define the communication type between different components, which has also been in other middleware communication systems before [16, p. 290]. These message interface definitions must be used and defined for each type of ROS2 communication primitives.

The most basic communication nodes used to share information is over **topics**. This communication method uses the **DDS** standard to send abstracted messages over the publisher-subscriber method [38] [3, p. 3]. This method means that a node sends information continuously to the ROS2 network, and multiple nodes could receive or subscribe to it. Since ROS2 utilizes DDS standards, all nodes can discover nodes they need without a centralized communication host, also called peer-to-peer communication [3, p. 3]. Generally, this communication scheme is widespread in different middleware systems [16]. At the same time, the components that publish this information often also can read other sources of the information [16, p. 290]. ROS2 publisher is usually responsible for defining when the data is available for other components in the system. This means that the publishing node can read information from the sensor and process it, and when this process is done, the information is broadcast to the system.

Due to the capabilities of the DDS enabling the **quality of service** settings, users can define the behavior of this publisher-subscriber method. For example, the user can specify the sensor data to be broadcast only once and never sent again after that. This allows the data to be transmitted fast in the network but does not account for a lossy connection that can prevent the subscriber from receiving anything. Similarly, the QoS settings can be defined so that the broadcasted messages are re-transmitted if some packets have been lost during the transmission [39]. The downside of QoS settings is that the target participants in the communication networks must have compatible QoS settings enabled to enable communication between them [39].

4.1.2 Services and Actions

There are scenarios where a node could need information or processing from another node, and it would not make sense to broadcast this information through the whole network. There is another interface called **service** which allows two-way communication between nodes [40][3, p. 3]. This interface is designed to implement one-time client-server or point-to-point communication where the client requests a server for some service to be executed [16, p. 290]. One service can have multiple client nodes that can request information when they require it. The communication model in services is simple. The client node sends the request over the service topic to the service node for requesting service with some prerequisite data if needed, after which the server will reply immediately when the service is done.

Since services were meant to implement one-time service API calls for simple actions and topics and are designed to provide a continuous flow of information from the node, there is the implementation for the third type of interface called to the **action server** [41][3, p. 3]. This server is designed to implement long-time operations like path following or trajectory execution. Similar to other communication methods, actions are also asynchronous. Figure 4.1 shows the communication architecture of the action server and client.

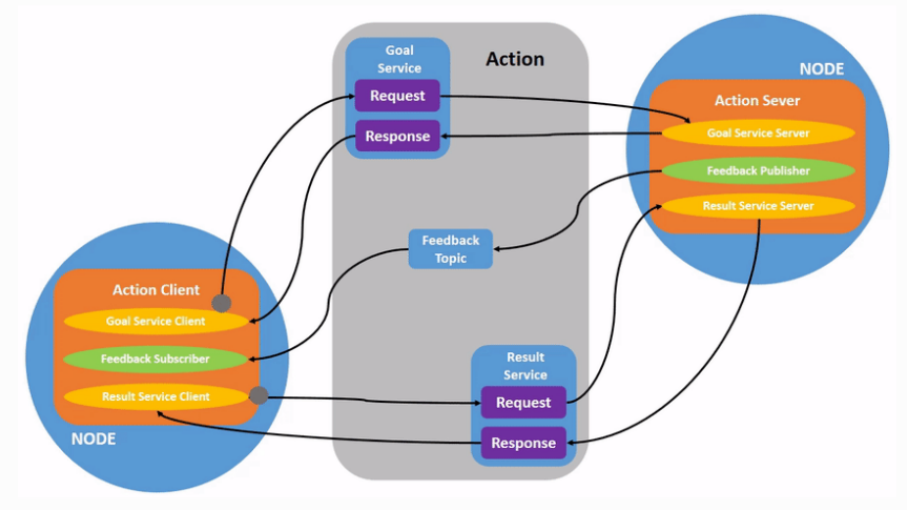

Figure 4.1. *"Understanding Actions" by Open Robotics is licensed under CC BY 4.0. [41]*

Actions are built upon three different states: goal, feedback, and results. If we inspect actions servers closer, we can see that it contains a service call for starting action, a publisher to send feedback to the client during the action, and a service call to get results from actions. As with service, actions require the user to define an abstract message interface that consists of the three previously mentioned states. Action servers, for example, initiate longer actions in the system that are wanted to be executed. This could mean path following. In the path following mission, the message definition could be the path that is wanted to be followed, feedback could be the path progress in meters or percentiles, and the goal be if the path following were successful.

4.2 Developer tools

The second of the earlier mentioned categories that ROS2 offers for developers and users is developer tools. There are many tools in this category, but the most notable are different tools for simulation, visualization, package management and building, debugging, launch, and configuration tools [3].

Simulation tools are essential when making large robotics applications since the simulation can considered a proving ground for the developed algorithm or application before implementing it with the real machine [42]. Developing the application in the simulated environment offers other benefits like generating data sets fast with different algorithms, a safe and controllable environment where nothing can break, and the possibility of reduced development and design lifecycle [42]. Gazebo offers a simulation environment to simulate the physics and kinematics of the robot, sensors, and environments [43]. The out-of-the-box integration with ROS2 with controllers and sensor simulations offers a way to make simulated machines that act similarly to real machines. In this thesis, the gazebo was also used to develop the base system and verify the concept before moving to the

actual machine.

On the other hand, Rviz2 (ROS visualization 2) can be used to visualize and see the underlying states of the system and visualize different objects, maps, and sensor data for the system either from the actual or simulated robot [44]. Visualizing the sensor data like lidar (Light Detection and Ranging) point clouds, generated paths, and the robot is usually needed. This all can be done in the Rviz2. These integrated tools save time by enabling the debugging of the system and testing immediately without the need to develop software for each application.

4.3 Frameworks

As mentioned before, ROS2 could be divided into three different categories, and algorithms were one of these categories. ROS2 has many frameworks that try to answer general problems that many robot systems have introduced in the form of questions in the chapter 1.

Frameworks in this section try to answer these problems. Even though many different frameworks try to answer the same questions in various manners, the ones selected to be introduced in this thesis were utilized at some level to generate some parts of the target machine architecture or used to generate parts of the pallet-picking application. Many of these architectures are designed to be general and applicable to many robot models.

4.3.1 TF2 and transformation trees

Transformation trees are an essential part of robotics applications. This importance comes from the need to know the correlation between different joints and sensors compared to each other. This information can be utilized in many applications, but the most notable ones using these transformations are control algorithms for joints and sensor fusion algorithms for sensors. This section will introduce the ROS2 tool for easily keeping track of these transformations.

This is where the ROS2 geometric transformation system called TF2 steps into the picture. This library is designed to calculate these geometric correlations between the tracked frames inside the system in question [45, p. 65]. These frame transformations are usually required in every robotics system and must be implemented using some linear algebra library. By utilizing this library, users can avoid errors in self-made calculations and avoid writing complex frame-tracking systems [7]. This library does not create any complexity to the previously defined interfaces that belong to the basic concept of ROS2. They use the simple publisher-subscriber scheme to keep track of frames using specialized TFMessage that users can use to populate the transformation trees in their system.

In ROS2, TF2 transformations are divided into two different subcategories called dynamic and static transformations [7][45, p. 66]. As the category names imply, these differ in a way that dynamic transformations can change over time, and static ones remain the same over time. Static transformation assumes that the static vertices exist and are published only once in the system. This can make the load in the transformation tree lighter since there are fewer transformations to keep track of and update while listening to all of the frames in the system. Then, Dynamic transformations are designed to keep track of moving parts in the system like manipulator joint angles[45, p. 66].

A transformation tree can be expressed as a graph. In this graph, all tracked coordinate frames can be considered nodes or vertices, and the geometric correlation is the edge of the link between those nodes. Due to the need for the transformation tree to be a fast operation when calculating the correlation between two connected nodes, the graph has been limited to the tree, or an undirected graph has been decided when designing the TF2 library [7]. This means that each node can have multiple child nodes, but there must be only one parent for each node.

When there is a need to calculate the correlation between two frames or the current frame's position in another, TF2 provides an easy API to calculate these. This not only calculates the frames with the last known correlation but also the library can calculate the transformations at some point in time and interpolate between known states to find as accurate a transformation as possible at any given time frame.

To better understand the transformation trees in the ROS2 application, we can take an example application with these sensor dependencies on the robot's base and different joints that can move. The figure 4.2 shows a simple Turtlebot3 transformation tree.

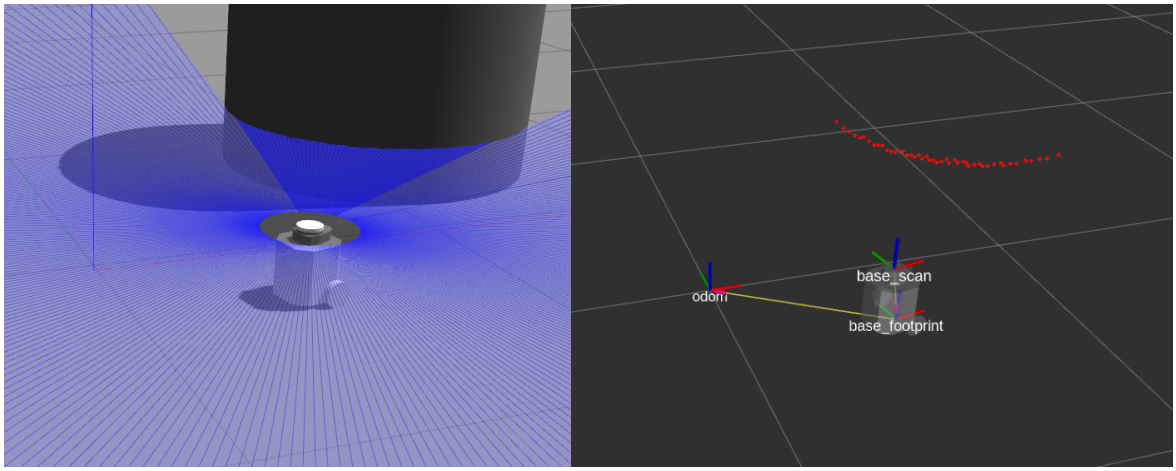

Figure 4.2. *Simulated robot in gazebo and Rviz2 showing the transformation tree frames in from simple Turtlebot3, showing "odom" and "lidar_scan" frames about the "base_footprint" frame of the robot*

The figure shows a simulated Turtlebot3 burger in a gazebo environment and part of its

transformation tree in the Rviz2 environment [46]. To demonstrate the transformation trees, it is necessary that even a tiny robot with a small number of joints can have multiple joints and sensors to track. In the figure, To transform the data from each sensor to the robot base frame like the 2D scan of the lidar, it is necessary to know the link between the robot base and the lidar, which usually are static transformations in the rigid robot body; hence, they are also static transformations in the transformation tree. The joint states are dynamic transformations in the transformation tree since these can update very fast depending on the sampling rate from the encoders on the wheels. Other transforms do not necessarily exist on the robot's body but tell where the robot is located compared to its starting position in the world and where the robot is in the world. In the image, the axis called "odom" shows where the robot started moving and the displacement between the robot's body and the starting position of the movement.

This library has been used heavily in this work context to continuously transform frames and paths from one frame to another and keep track of the system states during the pallet-picking application. Although these functionalities could have been implemented by hand, it is less error-prone to use verified working libraries for this.

4.3.2 Robot localization

Robot localization is a framework for ROS2, designed to combine multiple sources of sensor data and provide stable 3D localization for mobile robots. This package aims to answer the previously mentioned question "Where am I?", which is critical to answering the other two questions. Robot localization can be divided into three different parts: ekf_localization, ukf_localization, and navsat_transform [4]. Robot localization uses different Kalman filter implementations to fuse multiple sources of data, such as extended Kalman filter (EKF) or unscented Kalman filter (UKF) [4]. The extended Kalman filter is a filter for estimating nonlinear systems by using several measurements and statistical noise to generate an estimate of the state [4].

The extended Kalman filter consists of two steps: prediction and update phases. During the prediction phase, the filter integrates the state of the estimation with the given sensor input to the system with a nonlinear state transition function in the equation. In the robot localization package, the motion model is an omnidirectional motion model. Estimation also has some uncertainty or covariance. During the prediction phase, the uncertainty of the estimation gets worse over time. To mitigate the uncertainty, the filter has the update phase, where the correction data from different sensors can feed the correction to the filter. This update comes from the measure or sensor that can measure the state, like the GNSS unit. The update phase corrects the filter estimation while decreasing the uncertainty.

Previously mentioned TF2 has also been used in the robot localization package. This

package transforms different sources of information into the target frame for localization. For example, GNSS information from the GNSS receiver needs to be transformed to the frame the system is localizing to avoid introducing additional errors to the localization. The second utilization of the TF2 is to represent the localization in two frames if the system wants to be fed update phase data: map and odom [47]. This formulation divides the localization processing into two filters: local and global. The local filter continuously calculates the movement from the starting position, utilizing only the prediction phase of the EKF filter. The local filter will drift over time. Global filter estimates of the transformation from map frame to base link while utilizing the prediction and update phases to provide accurate estimation that does not drift, but this estimation is prone to "jumping" discretely. Although the global filter estimates the map's transformation to the base link, it is applied to the map's transformation to odom frame. This means that the starting place of the machine is moved during the navigation to keep the global estimation accurate while allowing the local estimation to be continuous without the previously mentioned jumps in the estimation. This way of calculating the localization in two frames is defined in the design documents of ROS enhancement proposals (REP) [48].

4.3.3 Navigation2

Navigation2 stack is, as the name suggests, a framework that implements navigation and control stack for holonomic and nonholonomic robots [5]. Navigation2 stack aims to answer "How do I get there?" by generating and following paths. Navigation2 stack components are composed of multiple ROS2 servers that implement different functionalities like planning, control, and recovery [5]. Navigation2 aims to provide safe and reliable collision-free indoor and outdoor navigation [5]. Every server has been implemented as a plugin to make the system configurable and modular. Each server can be configured to use multiple plugins for different robot types. Navigation2 also provides sensor information processing and saving in global and local costmaps that keep track of the static and dynamic obstacles that local and global planners avoid. For the control of the information flow and task management, behavior trees have also been as task-level management tools in the architecture [5]. Navigation2 stack supports many kinds of robot models, most notably differential, omnidirectional, legged, and Ackermann type of machines [49].

Navigation2 utilizes heavily other algorithms to enable the integration to the user's robot applications. Navigation2 recommends users use previously mentioned existing ROS2 packages like robot localization and TF2 to provide fast integration to their system. There are other tools that should be used during the integration process, like the SLAM toolbox, which is a toolbox for mapping the environment.

The most notable restriction of the navigation2 stack is that the planning is restricted in the 2D plane. This stack cannot be used for the machine's manipulator without major

modifications to the base system and may not be sufficient navigation in uneven terrains. Although many robot models were supported in the library, it does not support some common types in heavy-duty mobile machines like articulated frame steering [50], which in this target system could mean non-optimal path generation and path following.

Although in this application, there could have been heavy utilization of the navigation2 stack from the behavior trees to the control, There was a decision to implement the behavior tree stack from the beginning and only use the navigation2 stack to generate obstacle-free paths for the functional layer of the control system. Since the mapping had been done already and an existing localization system had been implemented, there was no need to use any other parts from navigation2 other than the planners and costmap tools.

4.3.4 Moveit2

Navigation2 was trying to solve the planning and control problem in the mobile robot platforms, but in our case, we also have a manipulator attached to the HDMM. A robotics manipulation platform for ROS2 exists called moveIt2. MoveIt2 solves general problems in the robotics manipulation field, including kinematics, motion planning, perception, and control [51][6]. The platform itself is robot-agnostic and uses the described kinematic chains to perform different calculations for motion planning and kinematic control, utilizing many other libraries to implement kinematics, collision, and planning such as KDL (Kinematics and Dynamics Library), FCL (Fast Collision Library), and OMPL (The Open Motion Planning Library) [51][6].

Similarly to navigation2, moveIt2 is also implemented as a plugin-based architecture so that the system components, like motion planners, can be changed easily. The functionality of MoveIt2 centralizes to the move_group node, which implements interfaces to the base functionalities of the platform for different APIs like C++, python, and ROS2 APIs and coordinates the execution of functionalities. The platform can make plans in joint space and Cartesian space to reach the target pose for the end-effector. Many of the internal planners of MoveIt2 utilize sampling-based planners that make the trajectories unique and sometimes unintuitive in each control iteration, even though the initial and goal pose stay the same [51].

To keep track of the environment, MoveIt2 also implements a Planning scene that tracks the relevant objects and obstacles in the environment based on static obstacles and perceived information. This data is later utilized in the trajectory generation to make an obstacle-free motion for the manipulator [51].

5. PALLET-PICKING SYSTEM FOR HEAVY-DUTY MOBILE MACHINE

An autonomous heavy-duty mobile machine system needs to perform autonomous pallet-picking, given the rough estimation of the pallet position in the mapped environment. While the initial position of the pallet is uncertain, the pallet should be picked without replanning. The target platform for this application was highly modified Avant 635 at Tampere University, which can be seen in Figure 5.1.

Figure 5.1. *Target machine for the pallet-picking application*

This target application leads to the following system features and requirements for the target application that are required for the machine to operate autonomously in the given task:

- Generate a plan to the given target pose.
  - The path shall be obstacle-free.
  - The path shall account for the physical constraints of the machine.

- Detect the pallet.
  - Detection shall be done with lidar.
  - Detection shall be represented in the system as a part of the transformation tree.
- Manipulator is used to pick and lift the pallet.
  - Given the target pose, there shall be an obstacle-free trajectory and movement to be done.
- System follows the plan to the given pose.
  - System shall stay on the path inside the acceptable limit.
  - The following unit shall provide feedback to the system from the status of the path following.
  - Given path shall be moved given the initial pallet pose was incorrect.
- System shall have state estimation and localization during the operation.
  - Localization shall be implemented with an extended Kalman filter with the following sensors: IMU (Inertial Measurement Unit), wheel odometry, and GNSS-RTK (Global Navigation Satellite System - Real Time Kinematic).
  - State estimation for the manipulator and articulated angle shall be done with IMUs and resolvers.
- System shall have common communication middleware between system components.
  - Low-level system communication shall be done with CAN (Control Area Network).
  - High-level communication shall be done with ROS2 common interfaces such as topics, services, and actions.

The target system can move autonomously with ROS2 as a base interface. This allows any ROS2 system architecture for manipulator control or path following utilizing the ROS2 to be integrated with the base architecture by defining the communication topics to match the base system. Given the machine operations, such as navigation and manipulation, and system features and requirements, the compound actions for the behavior tree are easy to define. Pallet-picking as an application can be broken down into at least the following high-level behaviors for the system, system components, and behavior tree nodes:

- Define the target pose of the pallet or drop-off location.
- Generate a path for the mobile machine.
- Follow the path.

- Generate a trajectory for the manipulator.
- Follow the trajectory for the manipulator.
- Recognize the target pallet.
- Reconfigure the transformation trees when the pallet is not in the correct spot.

To break down the mission into task definitions and coordinate with the base system, a behavior tree will be implemented to handle the system components' decision-making and coordination. The tree will act as a decision layer in the layered architecture where each compound action recognized will be tied into a behavior tree node that can interact with the base system. In this chapter, the behavior tree used to enable pallet-picking will be introduced, and how the system has been tied to the base functionality of the Avant machine will also be shown.

To enable the system to detect pallets, a solid-state lidar has been installed on the fork of the machine that publishes the point cloud from which the pallet is seen on the run time. This information is then applied to the transformation tree of the system using the TF2.

5.1 Pallet-picking as reactive layer problem

As stated in the item list in chapter 5, one of the required actions was to be able to reconfigure the transformation trees when the pallet is not in the correct position according to the base map and detection. This means that sometimes the pallet's target location is incorrect, and even a small deviation could lead to poor performance in pallet picking. In the real-world application, this could mean that the pallet location stored in the management system has moved due to some external effect, such as a human operator moving it, which causes uncertainty in the pallet location and initial position. This deviation and its solution can be divided into two different problems: deliberative-layer problems and reactive-layer problems. In the deliberative-layer problem, when the deviation of the pallet pose is detected, there is a need to detect the new pallet pose and plan a new path to it so that the machine can dock into the pockets of the pallet. The reactive problem again utilizes the state representation of the machine and pallet and shifts the path with the pallet pose, given the deviation from the estimation. This will lead the path following the algorithm to fix the error in the initial estimation. Path following implementation itself does not see the movement of the path. Still, it sees the change in the localization of the robot, which moves according to the transformation of the pallet location. The latter method naturally requires the system to detect pallets from far enough to have time to converge back to the path after the pallet is detected.

To handle the uncertain initial estimation of the pallet frame, we need to represent much of the required frame information, such as the base of the machine and paths in the pallet frame, during the navigation. This representation allows moving the pallet frame in the

map coordinate system while allowing moving of the path without replanning during the navigation. To understand this, we need to define some of the transformations in the base system and show what is represented in what frame. Figure 5.2 shows some of the relevant transformations for the navigation problem.

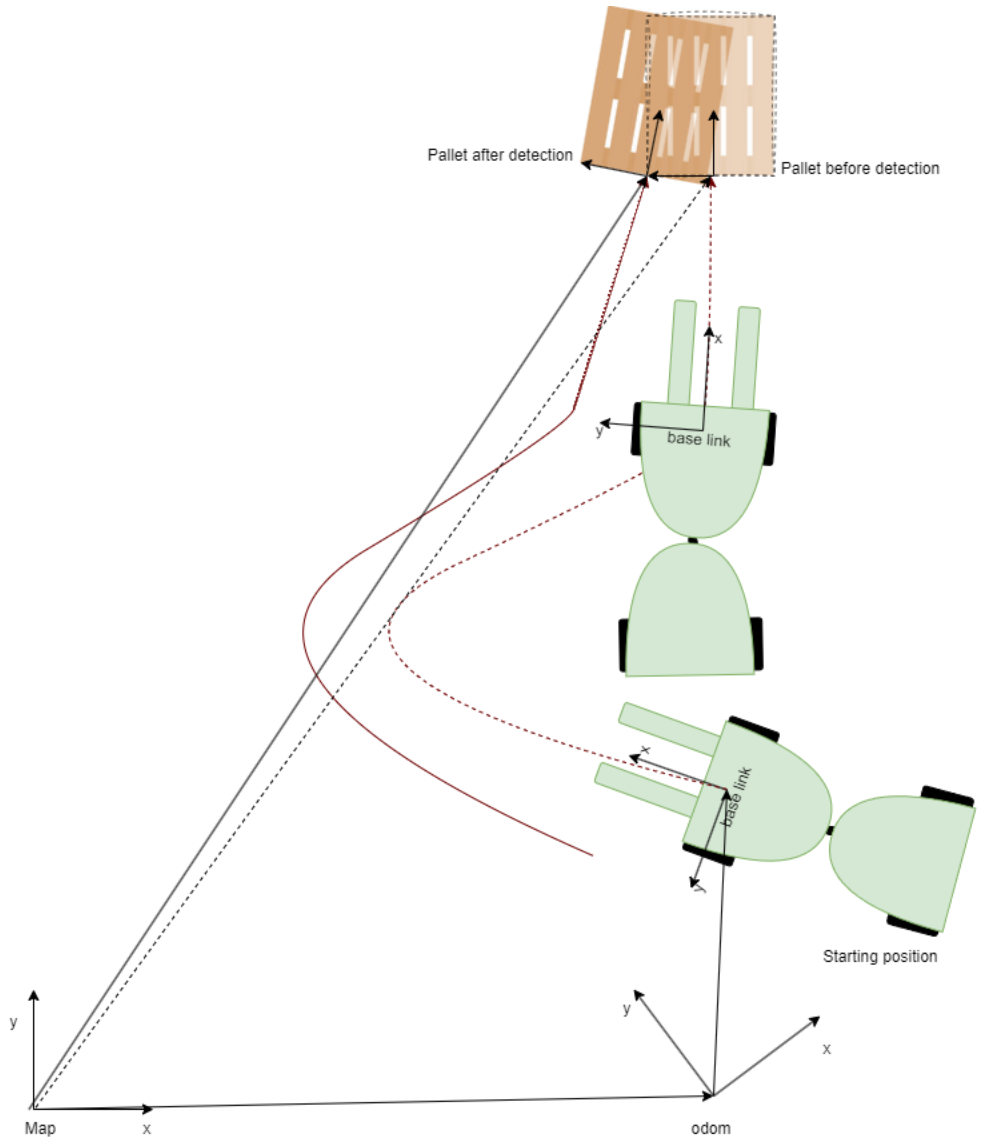

Figure 5.2. *Representation of shifting the pallet and path in the map frame after the detection of the pallet*

The figure shows that the root transformation for the system is the ”map” frame, which has children tree nodes ”pallet” and ”odom”. The pallet frame is the position of the pallet’s front side in the map coordinate system. Similarly, the odom frame represents the motion in the map frame from the starting position of the machine. The transparent pallet represents the pallet location in the ”map” frame after the user inputs it into the system. Frame odom represents where the machine started operation after the turn-on. Traditionally operation or path following the robot’s displacement or odometry is either represented in the map

frame or the odometry frame. In this application, though, the machine base link frame odometry is represented in the pallet frame. The position of the machine in the map of the environment can be represented in the following form in the equation 5.1

$$^{map}\xi_{base} = {}^{map}\xi_{odom}\,{}^{odom}\xi_{base}. \tag{5.1}$$

The manipulator end effector state takes the representation from the equation 5.2

$$^{base}\xi_{forks} = {}^{base}\xi_{boom}\,{}^{boom}\xi_{telescope}\,{}^{telescope}\xi_{forks}. \tag{5.2}$$

and the complete state estimation in the pallet frame follows the form in the equation 5.3 where the $^{map}\xi_{pallet}$ is the transformation of the pallet in the map frame

$$^{pallet}\xi_{forks} = {}^{map}\xi_{pallet}\,{}^{map}\xi_{base}\,{}^{base}\xi_{forks}. \tag{5.3}$$

Since the path planning is generated based on the map frame, we need to first transform the path from the global map frame to the pallet frame so that the final point in the path is at the origin of the pallet frame when the machine detects that the pallet is not in the given goal position it sends a request to move the pallet to the correct position according to its localization in the map frame. The position fix of the pallet follows the following form in the equation 5.4, where the ξ is the homogeneous transformation matrix.

$$^{map}\xi_{pallet} = {}^{map}\xi_{base}\,{}^{base}\xi_{forks}\,{}^{forks}\xi_{lidar}\,{}^{lidar}\xi_{pallet}. \tag{5.4}$$

When the pallet frame is shifted, it also automatically corrects the path representation and the machine representation to the correct locations since these are referenced in the pallet frame. This can be seen in figure 5.2 where the path continuous line and pallet have been shifted from the dotted lined pallet and path after the pallet detection is complete. After this sift, the machine is not longer on the path to the path following and has to minimize the error to the target goal since this pallet movement is now handled as a shift in the localization according to the odometry. In this application, it is assumed that the pallet is detected early enough that the machine has time to minimize the error in the path before the pallet for successful docking to the pallet.

5.2 Distributed control system for autonomous HDMM

Combining the base implementation of the autonomous HDMM and the behavior tree implementation to the distributed control system, it is divided into three different PC units. Communication between the PCs and internal components is handled with ROS2, which

allows peer-to-peer communication between the machines. The system load is divided according to the figure 5.3

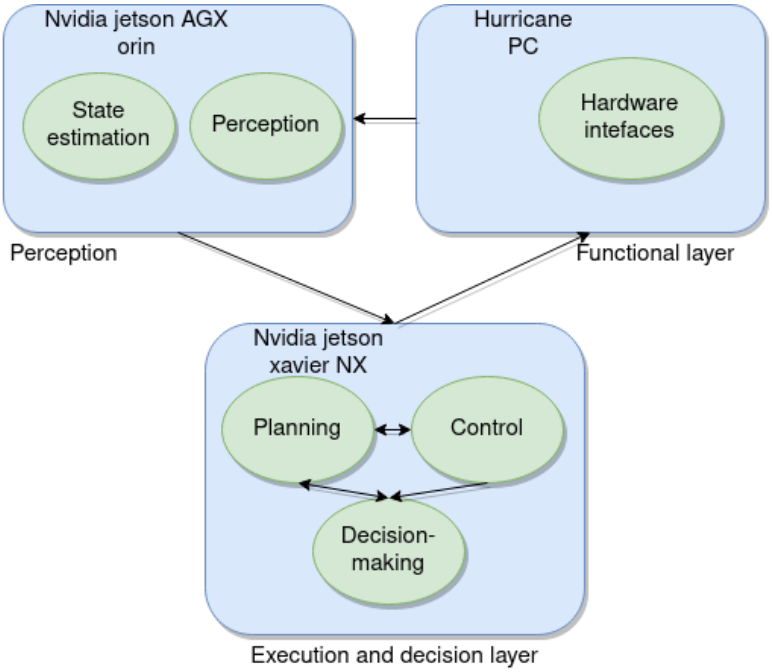

Figure 5.3. *Architecture division in the physical PC setup in target platform*

The system load is divided into three different PCs that all have different purposes in the architecture. Hurricane PC is responsible for parsing the sensor data and abstract ROS2 messages so that they are routed to the target platform correctly during the task execution. Nvidia Jetson AGX orin handles the state estimation, localization, and perception during the runtime and passes this information to Nvidia Jetson Xavier NX for the decision-making in the BT, and planning in the executive layer algorithms.

5.3 Behavior tree for pallet-picking using ROS2

From the architectural point of view, there are a couple of ways to implement the behavior trees into the base system architecture, using ROS2 and DDS as middleware for machine-to-machine communication. This means that when implementing the behavior trees, there can be very simple behavior tree nodes that just implement previously introduced ROS2 action and service clients and only interface with the underlying functional layers in the system. In the figure 5.4 can be seen both versions.

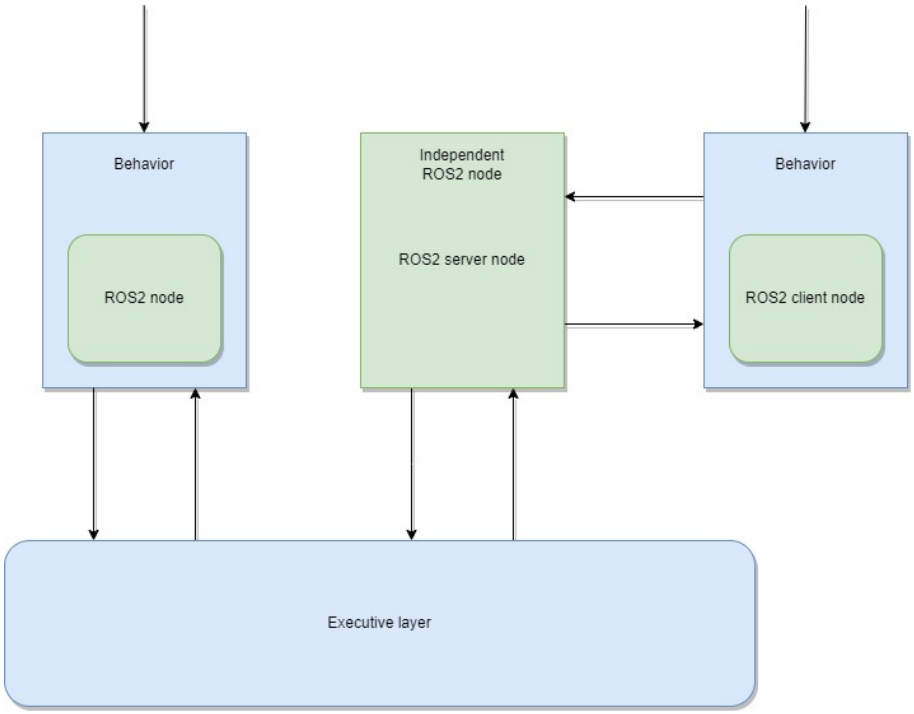

Figure 5.4. *Behavior trees component interacting in the layered architecture*

On the right side of the figure is a simple interface that only requests an underlying manipulator planner interface to plan a trajectory from the current pose to the target pose, and the behavior tree node would only provide the target pose. The second way would be to make the behaviors implement these underlying functionalities like trajectory generation, and following that, it would directly communicate with the executive layer, where the behavior tree action node would implement the functional layer of the system.

To integrate this implementation into the base system shown in figure 5.10, there was a need to connect the system to the existing interfaces of the system and create some interfaces that would provide certain interfaces that were not in the system yet. Figure 5.5 shows how the behavior tree binds to the ROS2 interfaces in the base system software stacks.

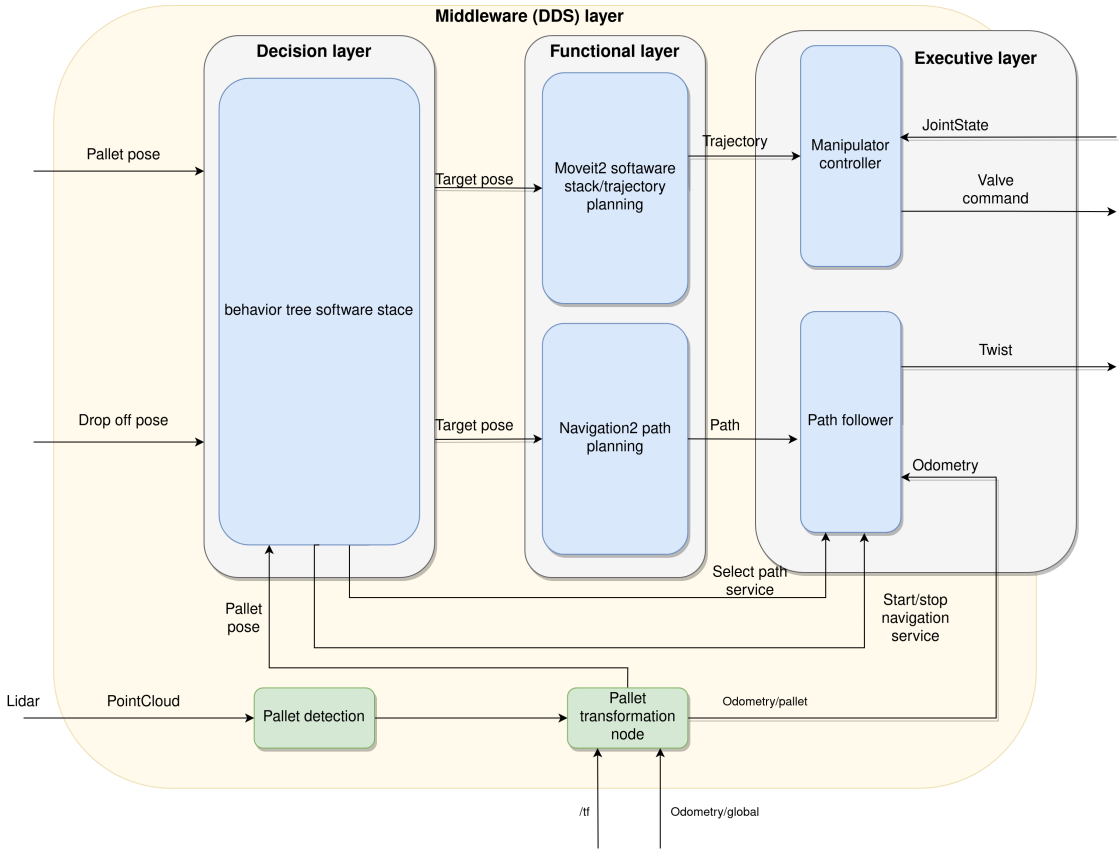

Figure 5.5. *Behavior tree integration to the base system architecture*

Due to the existence of the underlying components, interfaces, nature of the behavior trees, and ROS2's design of trying to increase the re-usability of the code, the first version was selected. This architecture also allows later integration of other tools and functionalities to the base system or integration of the behavior trees with other target systems. Also, the first variation would resemble more of the layered architecture where each layer in the component stack would have certain tasks in the hybrid system implementation, which also provides increased modularity in case the system needs to be increased. Odometry for the path follower is also transformed. Odometry is provided in the pallet frame, which is constructed from the global odometry, detection from the pallet detector, and the information in the current transformation tree, which is estimated in the machine fork frame.

Some of the introduced requirements are implemented as behavior tree nodes, but the features are hidden in the base architecture as active ROS2 nodes that provide information to the base system. Since behavior trees are designed to work as a branching tree format, we need some way to logically read this visually constructed tree in the figure 5.6. This means that reading the tree starts from the root node in the top left corner of the tree. In the navigation architecture, the execution of the tree starts only after the pallet picking pose and the placement pose have been fed to the system via ROS2 topics. While testing

the software stack, the target location of the pallet was placed into the Rviz2 visualization of the environment map beforehand.

Some requirements for the pallet picking application were stated earlier at the beginning of the chapter 5. To fulfill these requirements, there was implemented a behavior tree implementation that can be divided into logical subtrees in the following way in figures 5.6 5.7 5.8 5.9 to enable the system to navigate and pick the pallet autonomously. Figure 5.6 is the main tree in the structure that initializes the pallet position and moves the manipulator to such a pose that the lidar connected to the boom can observe the pallet location during the navigation.

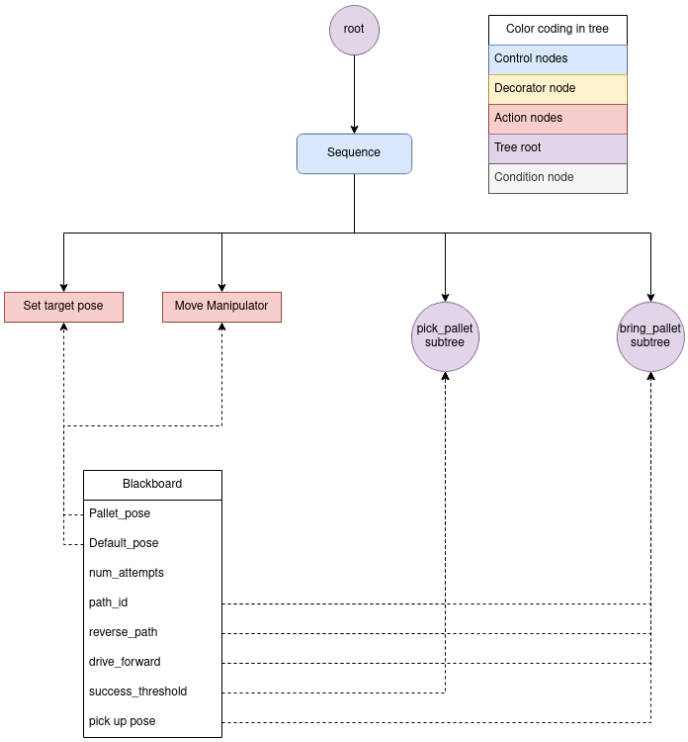

Figure 5.6. *Main Behavior tree designed in the work*

Figure 5.7 is the tree structure for navigating to the given pallet pose and picking it with the given detection pose. Figure also shows the color coding of the different tree nodes in the behavior tree structures used in the tree and subtrees. This includes generating a plan for the pallet pose, setting it to the path follower, and picking it up while navigating.

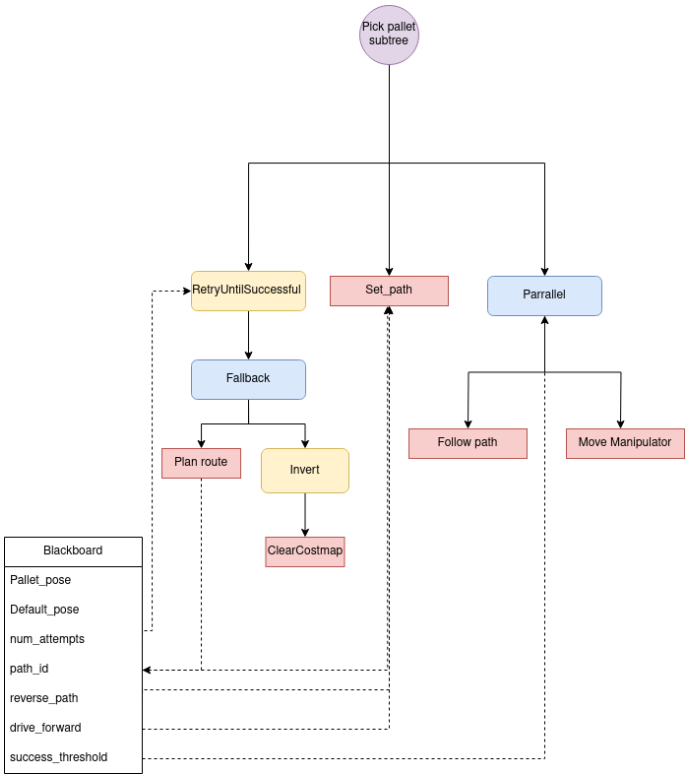

Figure 5.7. *Subtree for pallet-picking*

After successful pallet-picking action in the pick subtree, the machine has two options to initiate according to the subtree structures. Either take in the second pose given by the operator and move the pallet to the new location in the given environmental pose or move the pallet back to the starting position where the initial plan started. Figure 5.8 will take a new pose argument to move the pallet there and place the pallet on the ground.

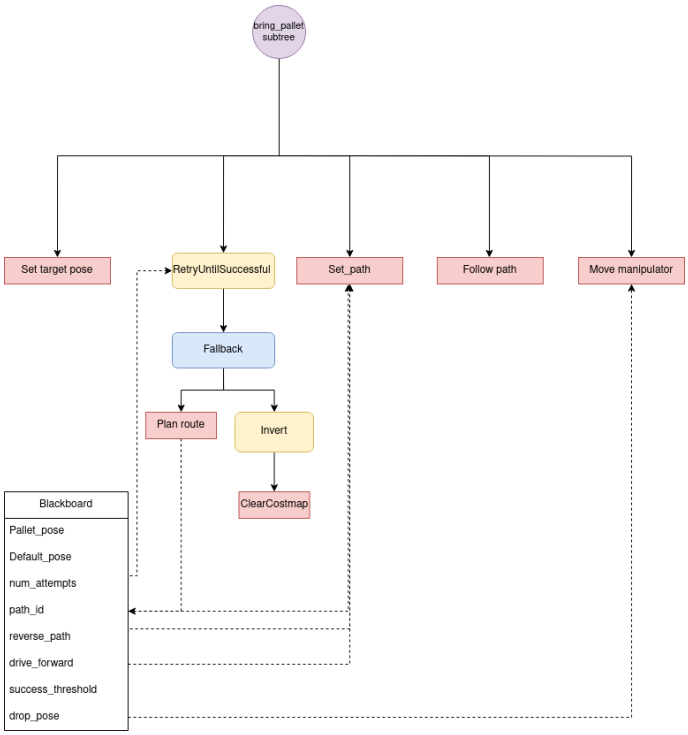

Figure 5.8. *Subtree for planning and moving the pallet to designated location*

Figure 5.9 will move the pallet back to the starting position and place the pallet on the ground.

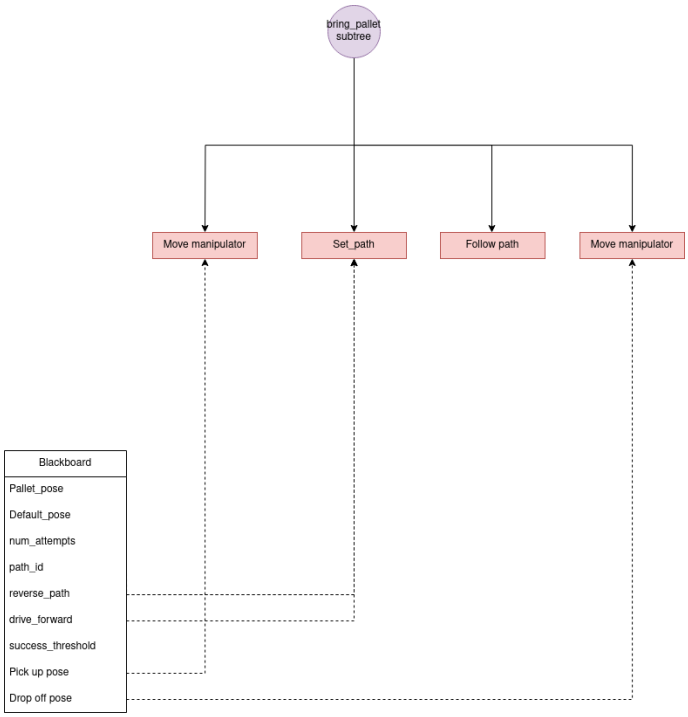

Figure 5.9. *Subtree for moving pallet back to starting position*

A couple of notes are to be taken in the trees: All blackboards in the images are the same

global blackboard that is initialized after the drop pose and pick poses for the pallet are assigned. If the planning fails in the navigation2 plan algorithm, there is a fallback function that tries to clear the cost map in the global or map frame, and this process is done the number of times that it is registered to the blackboard. Blackboard also distributes and handles many of the internal parameters in the tree structure so that there are fewer redundant ROS2 topics in the network that are broadcasted rarely.

This layering of the tree functionalities also allows the ROS2 architecture to follow the hybrid navigation architecture paradigm division. Path following could be extended to a reactive layered architecture where the reactive layer would follow figures 3.2 architecture containing an obstacle avoidance layer, where the velocity commands would flow in the ROS2 network. If, in the current implementation of the lower level control stack, there was something like software level emergency stop, it could be possible to implement new behavior and fallback to the main tree, which would tick the e-stop to active in case some major node would fail inside the tree logic. This behavior could be done if, for some reason, the path following deviates too much from the path, the pallet is not in the designed locations, or path generation is not possible.

Behavior tree implementation has one additional feature to enable the machine to initiate actions other than pallet picking. The trees implemented in the system are not static in the code. The trees are constructed on the runtime from, for example, a simple XML (Extensible Markup Language) file that states the node order and which node uses which kind of blackboard information. This means that in case of needing multiple different behaviors or tasks, the user could send a request to the behavior tree layer of the system to download a new behavior tree for the system in case the old tree is not suitable for the new task and all this could happen without the need to restart the whole system. Since the trees live as configuration files in the storage of the base system, it also means that many of the behaviors needed for new tasks can be constructed without the need to update the whole system architecture and components in the suitable atomic behaviors are implemented to the base implementation of the behavior tree already.

5.4 Architecture for autonomous heavy-duty mobile machine

The base ROS2 architecture of the implementation is in the target machine. The target machine was able to follow paths in the robot's map frame, create trajectories, and follow those for the manipulator in the base frame. Figure 5.10 shows the base system's information flow and relevant components. In the figure, the system has also been divided into functional, and executive layers. In the figure 5.5, the decision layer connects to the figures 5.10 functional layer. As can also be seen, these layers have been encapsulated by the middleware layer or the ROS2 RMW (ROS middleware) for DDS that handles the data transmission between the components and the layers. As the general in the layered

architectures in this one as well, the abstraction of the communication decreases as we get closer to the hardware layer where the high-level task definitions, such as trajectories or paths, turn into actuator-level commands, and the hardware commences these actions as a movement in the machine.

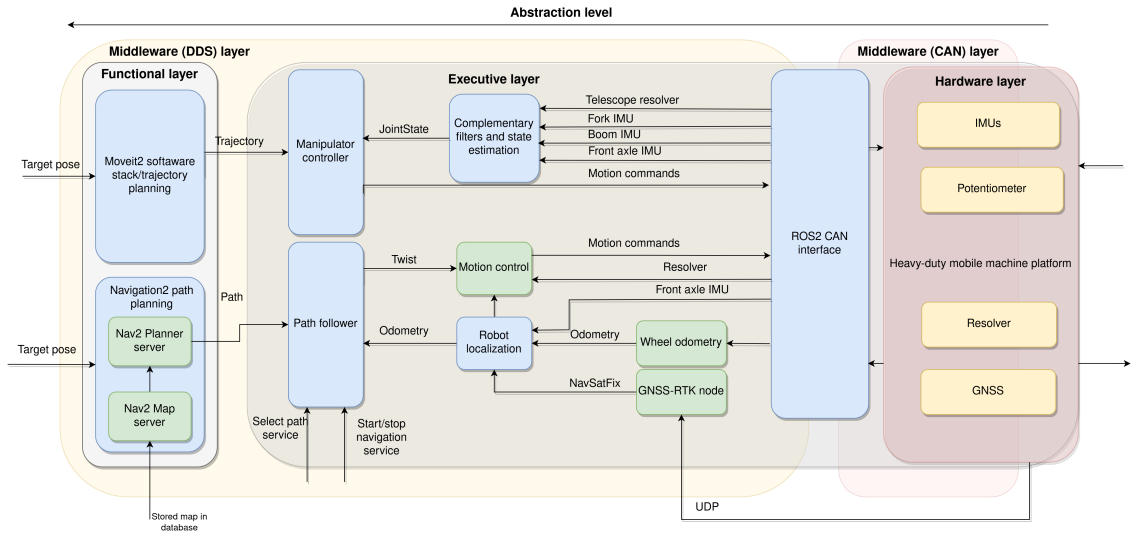

Figure 5.10. *Avant base navigation and control architecture with ROS2*

Figure 5.10 shows that the configuration requires many sensors to provide accurate state estimation for the manipulator and machine. Most sensors are integrated into the system through the CAN (Control Area Network) bus interface that communicates to the distributed base controller system. To enable ROS2 to be a viable option for navigation stacks in the target machine, there needs to be a standard interface for ROS2 topics and control interfaces. Most of the sensor interfaces are basic ROS2 interfaces providing corresponding information to the topic like ”Imu”, ”NavSatFix”, and ”Odometry”, which by state estimation nodes are turned to ”JointStates” and ”Odometry” messages in different frames to give the current state of the machine continuously to other components in the system. CAN interface also accepts valve commands for the hydraulics actuators, which are fed to the internal distributed control system of the machine to control the hydraulic actuators.

Many of the nodes in the architecture hide some of the required data and functionalities behind the dependency of the transformation tree data, which is distributed through the static transformations during the launch time of the system and through the dynamic interfaces for the center joint of the machine, state estimation, and localization. This data is then gathered and subscribed through the internal TF buffer in the nodes that require this information to look up or transform data from the transformation tree.

To provide functionality for the manipulator planning, an existing integration to the moveIt2 software stack allows motion planning and control for the manipulator. Motion control for

manipulation is the separate implementation integrated into moveit2 through the plugin interfaces, which allows the integration of the planning interfaces to the hydraulic actuator control of the machine. These base systems were utilized to implement the base functionalities or skills of the system for different manipulation tasks. Interfacing with the moveit2 stack is done through the moveit2 move group interface, which allows interfacing with the motion planning and control through the ROS2 service and actions [52].

The manipulator state estimation machine has a collection of state estimation nodes containing 2D and 3D complementary filters, estimating the orientation of the front axle, boom, and bucket, respectively, which are used to estimate the entire state of the end effector of the machine in the application. This information is also fed to the moveit2 framework to provide the initial state for the planning interface and feedback to the controller.

The figure shows the localization using the ROS2 robot localization stack. It accepts IMU, odometry, and GNSS data from the low-level ROS2 sensor drivers. In the base system, there is a continuous feed of sensor data fed into the localization system to provide accurate positioning. For the prediction phase of the filter, there is a feed from the wheel odometry and IMU for the system. To provide the update phase, there is a feed from dual antenna GNSS-RTK [53]. GNSS-RTK antennas can provide 1 cm accuracy in vertical and horizontal positioning [54]. It also provides an accurate heading for the system due to the dual antenna configuration. The GNSS positioning has been referenced to a local map of the environment to provide local transformation of the GNSS to the local area pivot point, which allows utilization of the prerecorded map for path generation in the environment.

The Navigation2 stack has been used with the environment map for obstacle-free path generation. The map contains the information of the static layer of the obstacles, which are then inflated with the nav2 inflation layer costmap to show unsafe regions next to obstacles for the planner [5]. Due to navigation2 not supporting the articulated frame steering type of machines, the planner assumes the system will follow Ackermann's steering geometry. The generated path is then given to the path follower software stack as a discrete path, which waits for permission to commence with the path following from the decision layer after it has been set for the target path.

6. EXPERIMENTS

In this chapter, the experiments will be concluded to measure the system performance in the pallet-picking application. Part of the system is to review how the target platform performs in the base actions coordinated into it since the pallet-picking, in this case, is highly dependant on the base systems functionalities like state estimation, path following, and manipulator trajectory following and the accuracy of each implementation. Naturally, this dependency on the base functionalities leads to the inspection of the behaviors of the components during the operation and pallet-picking application. While the system is dependent on these functionalities, many of them are coordinated by the behavior tree during the run time or are affected by the coordination.

While the accuracy of the base components is crucially important, so is the software and hardware performance. Even if the application can perform in the restricted test environment but not in the distributed control system of the platform, the task will fail. This leads to the evaluation of how the ROS2 and software stack of the system perform in the control system and if the given resources are enough to handle the communication and algorithms in the system. Many of the components have been implemented in Python, leading to the evaluation of the RCLPY (ROS Client library for Python), which is the Python implementation of the ROS2 that implements the simple interfaces for the nodes.

6.1 Pallet picking test setup

Tests include the application executing the defined application task of pallet picking in the test site. Very similar to the figure 6.1, there is a similar setup to the real world. Machine and target pallet starts from some predefined location in the local world coordinates. As stated before, there is a map of the target site the application is designed to work in. The location of the target machine is referred to in the localization of the map frame. After the setup, a command sequence needs to be sent to the behavior tree application from the Rviz2 view on the right in figure 6.1. Command sequence needs to define the rough position of the pallet in the map frame and where the pallet should be brought after the pick-up. After the behavior tree application accepts the command sequence, the execution of the application starts. This execution is then saved from the start to the pallet

picking using ROS2 bags that monitor the communication and the system states.

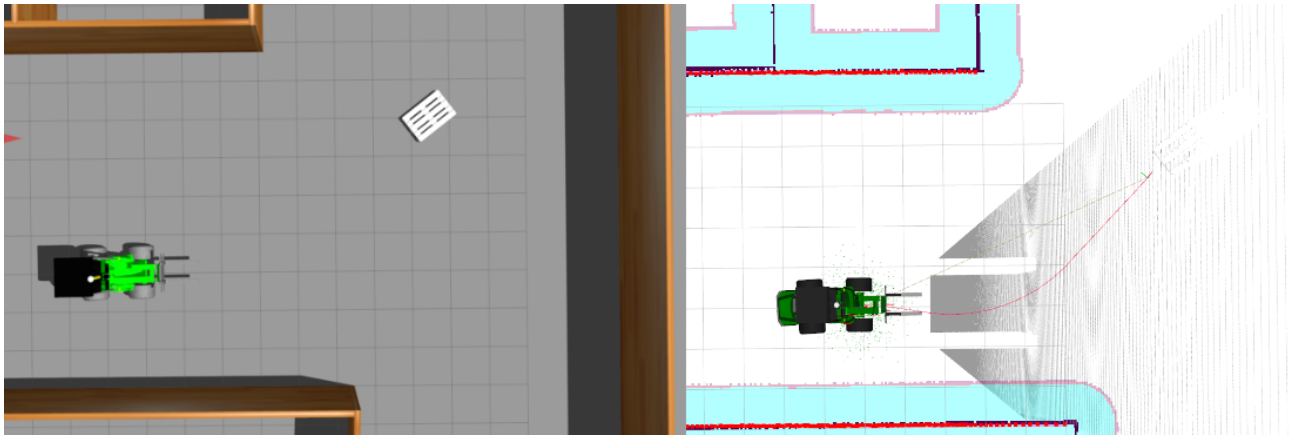

Figure 6.1. *Illustration of corresponding pallet-picking test setup in simulation*

Tests have been done in the mobile lab of the Tampere University using the target machine described in the figure 5.1. The target machine and the testing ground were provided by the Autonomous Mobile Machine Group at Tampere University. The whole system architecture had been divided into the three different PCs that can be seen in the table 6.1. During the pallet-picking tests, the software distribution in the hardware followed the figure 5.3.

6.1.1 Pallet-picking simulation results

The system was first tested in a simulated environment with a simulated version of the machine and environment. These tests included only functional tests to verify that the realized system architecture can converge to the pallet given the uncertain initial estimation of the pallet location and the above behavior tree implementation in figure 5.6.

The simulation architecture is similar to the real machine introduced in the chapter 5. The expectation for the real machine is that the hardware layer and interfaces have been replaced with the simulated environment and machine in the Gazebo with the same ROS2 interfaces.

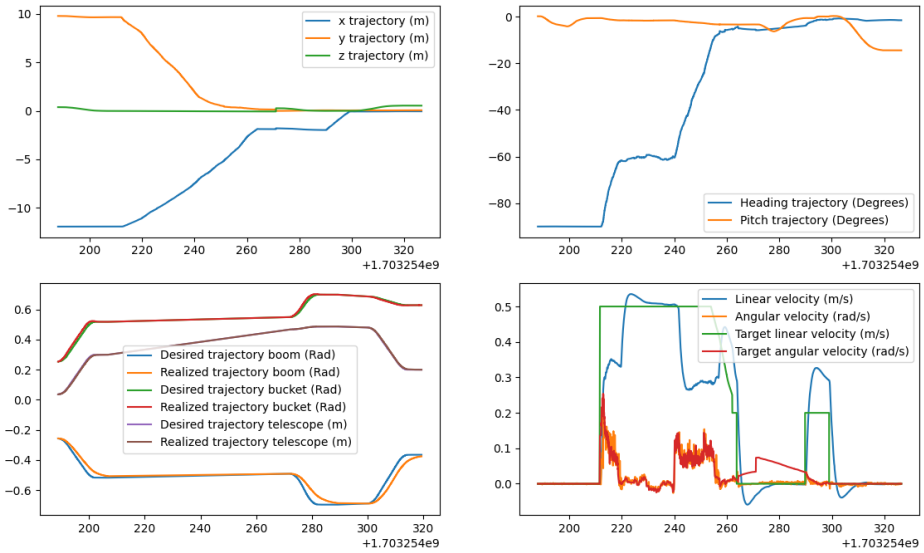

Figure 6.2. *Pallet picking results in the simulation environment*

The top left corner image shows the location of the manipulator fork in the pallet frame. The top right shows the heading and pitch in the pallet frame for the forks. The roll is not followed since it is not controllable degree of freedom in the machine. The trajectory of the moveit2 framework after the detection and lifting is in the lower left image. Finally, the system base twist command and response to it are in the lower right image.

Figure 6.2 shows that the machine can follow the input velocity responses from the path follower application. The pallet-picking system's reactive layer successfully fixes the pallet position estimation, which leads to a small jump in the localization seen in the top left image at roughly. At this time, the machine recognized the pallet location and fixed the transformation of it in the map frame. These results lead to the system being able to converge to the pallet, which leads to the BT formulation being able to commence the pallet-picking application in the real machine tests.

6.1.2 Pallet-picking real machine results

Results contain the data collected from the successful pallet-picking task missions as described in the beginning experiments chapter 6. In the review, there are key points for the application under review and their performance under the pallet-picking application. The application's key points for successful pallet picking are accurate path following, localization, state estimation, and manipulator control. Here are the results of four different successful pallet-picking mission tasks. The machine and pallet started from roughly the same location each time, and the location estimation of the pallet was set roughly to the location of the real pallet in the Rviz2 map of the area map. This test composition is

similar to the simulation tests, where these measurements are certain.

Figure 6.3 shows how the position error in the pallet frame converges to zero in all axes while the motion is active.

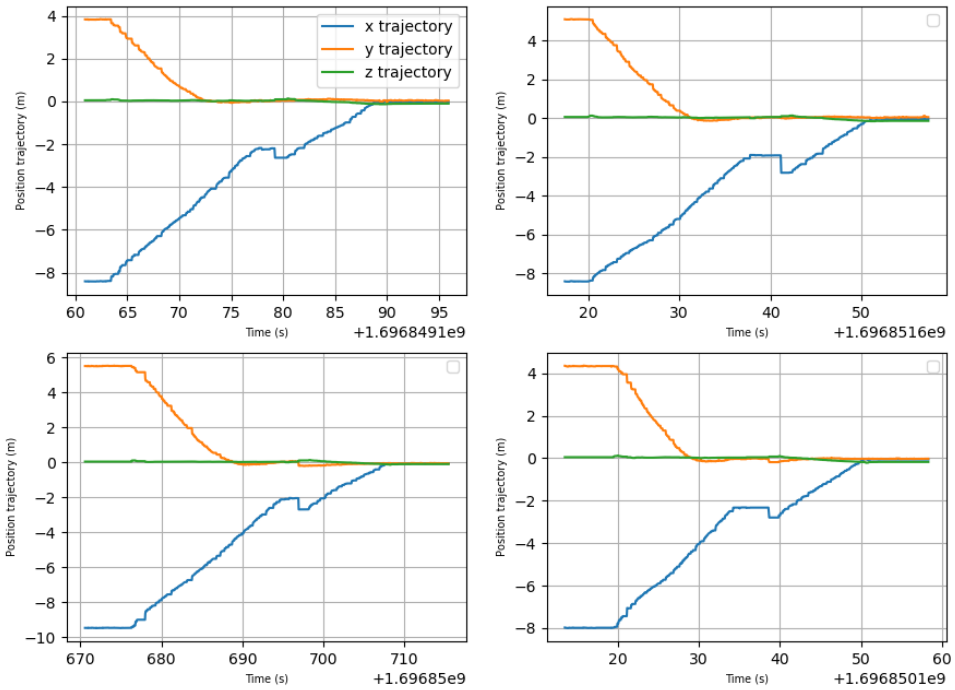

Figure 6.3. *Machine fork position trajectory in the pallet frame*

Since the error is plotted in the pallet frame from the manipulator frame, we can see the rough time of when the pallet location is fixed in the map frame. This happens roughly 2.5 m before the machine is at the pallet, which can be seen as a jump in the error. In these application cases, the y-direction jump was roughly 0.2 m. z-axis error is small due to the trajectory that the first action in the behavior tree was to move the manipulator to such a pose that the lidar connected to it can see the pallet in the ground, which happens to be very near to the pick-up pose of the pallet. Figure 6.4 shows the heading and pitch error in the pallet frame.

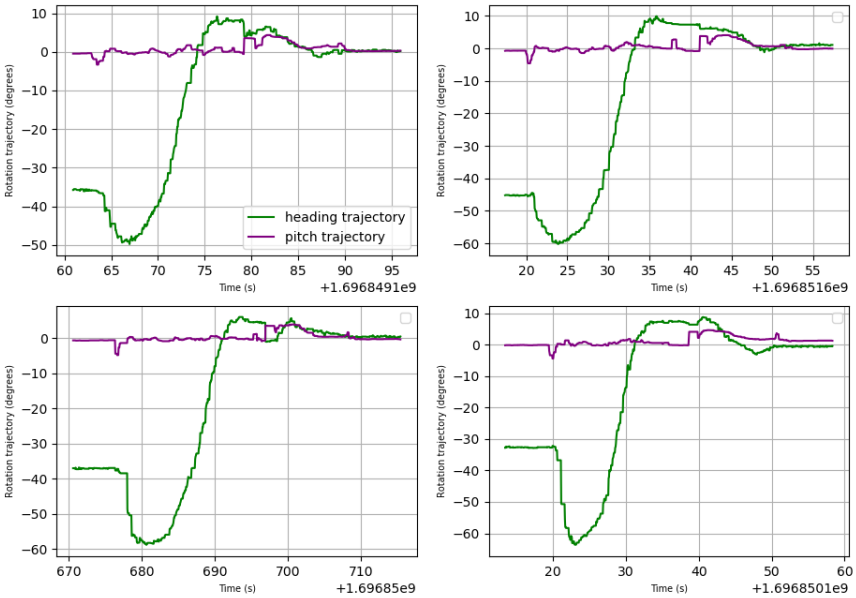

Figure 6.4. *Machine fork rotation trajectory in the pallet frame*

From figure 6.3 and 6.4 can be seen that the position and rotations converge before the pallet is successfully picked up, this means that the machine forks have been successfully docked to the sockets of the pallet. In these attempts, the initial estimation of the pallets' location was very good, given that the position and rotations do not change aggressively. Given the situation where the initial estimation was poor, and the pallet was detected very late, there will be a need for replanning of the path so that the system can respond to the issue that there is no time to converge back to the path before the aligning with the pallet. In figure 6.5 can be seen the velocity inputs from the controller and the system response after reaching the slowdown limit before the control limit is achieved during the slowdown period.

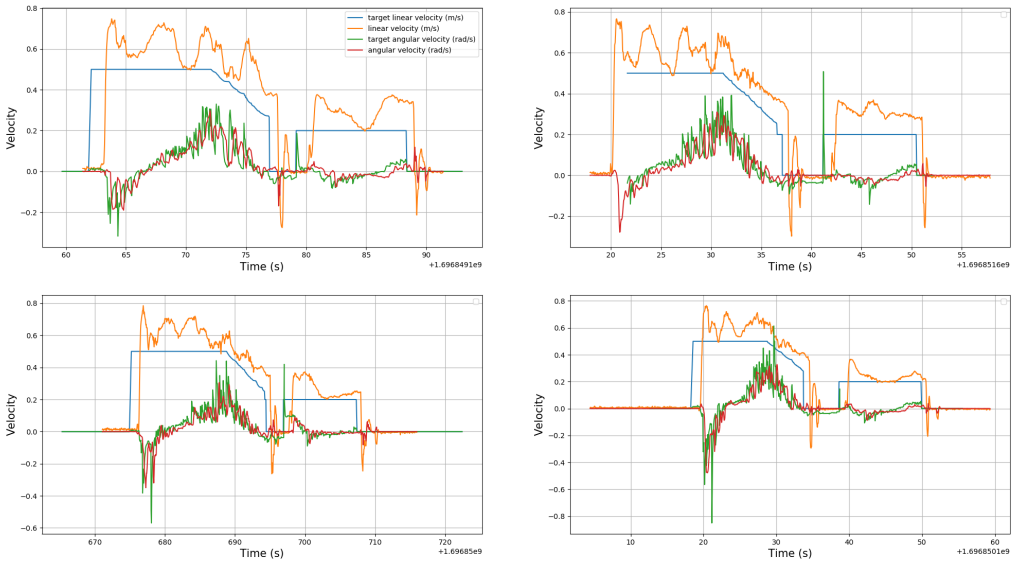

Figure 6.5. *Machine front axis velocity commands and response*

This includes the commands to the steering and velocity motion control unit without the manipulator control. As can be seen, the linear velocity input has been set as a ramp function to the system, and the open loop control response overshoots the set value and oscillates above it. The path following the controller has been tuned to minimize the lateral error very aggressively, which causes the angular velocity to oscillate too aggressively in the tests. In this case, the control and response of the steering commands follow each other very well, but the target velocity oscillates during the path following the curved path. Figure 6.6 shows the manipulator trajectory following during the operation. Since the planning and control of the manipulator is done in the joint space of the manipulator, the jump of the pallet reconfiguration cannot be seen in the machine's own configuration space, but rather, the planning architecture receives the pallet pose in the estimation of the machine base link.

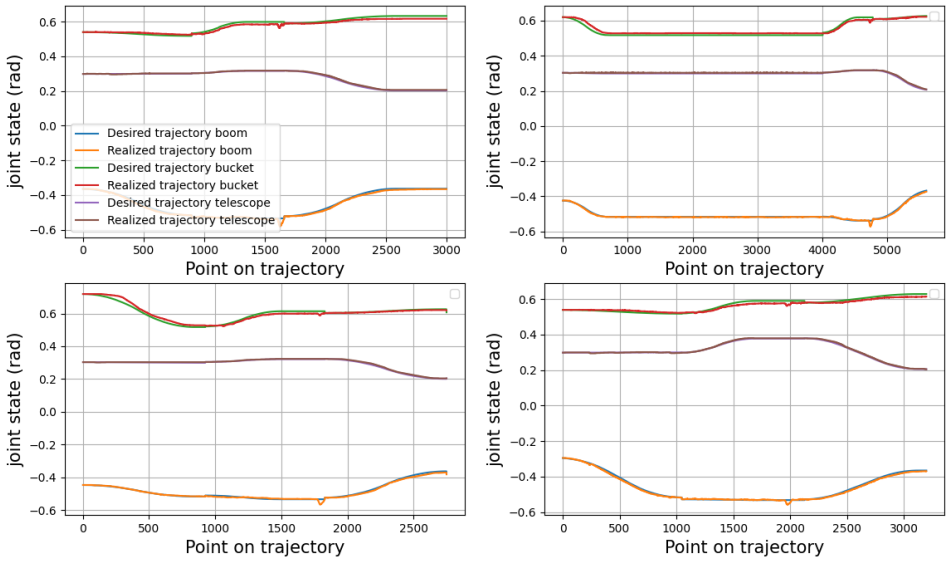

Figure 6.6. *Machine manipulator joint trajectory following*

At the beginning of the graph, the manipulator is lowered to the pose where the pallet detection is possible, and when the pallet is detected, the manipulator is then corrected to the pose of the pallet. After the docking has been successfully executed, the pallet manipulator lifts the pallet. Similarly to the velocity response figure 6.5 in both figures, it can be seen that the sudden stop of the machine affects the state estimation of both manipulator states and also in the velocity estimation, which can be seen as a sudden spike in the estimation due to the sudden stop of the machine.

In the end, the pallet picking application can pick up the pallet using the behavior tree structure in figure 5.6 given that the initial estimation was close enough to the correct pallet position so that there is no need for replanning of the path. This would be rather simple to implement into the tree structure if needed. There is a condition node that checks that if the pallets' distance to the path is within the given limit, we would not need replanning. Optionally, there could be continuous replanning in the system, but this could again assert additional computational load to the system during the new path generation loop. As in the simulation tests, the real machine tests verify that this composition of the ROS2 combined with the BTs in the layered architecture formulation with the decision-making or coordination layer can execute pallet-picking tasks, with certain limitations given the implementation of the BT.

6.2 ROS2 performance overhead test setup

This section handles the performance evaluation of the ROS2 in the application. This evaluation handles mostly the Python implementation of the RCLPY since most of the

nodes in the system architecture are implemented in Python, excluding more time-critical and high-rate nodes like the CAN interface of the system. During the testing of the architecture, there was a notice that RCLPY used a lot more resources than expected, seemingly without any reason, mainly CPU (Central Processing Unit) usage. The reason for the high usage is due to the investigation here to find what causes this high CPU usage when the RCLPY implementation of the ROS2 can be used in robotics applications and if there is additional overhead exerted to the system to other units, such as RAM (Random Access Memory). In the figure 6.7 can be seen the received topic rate of a couple of topics and the CPU load of the processor during the pallet-picking application and data recording.

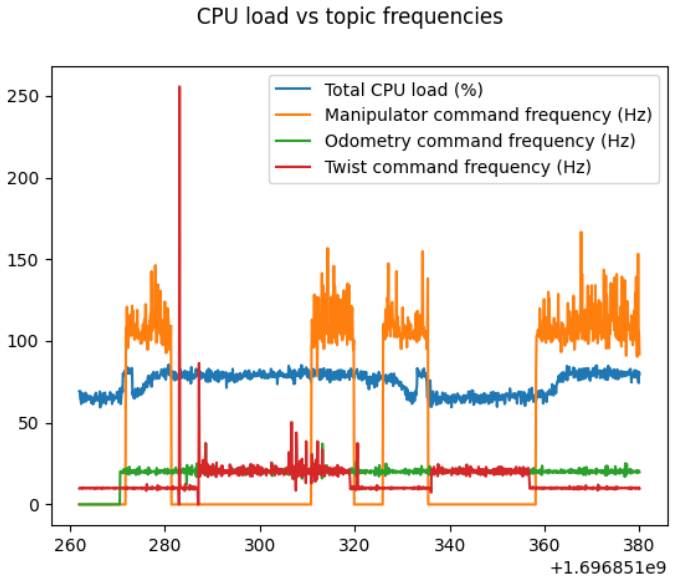

Figure 6.7. *Nvidia Jetson AGX Orin average topic receive rate and CPU load*

As could be seen from the image 5.3, this PC unit is responsible for receiving sensor data and doing state estimation and perception. This unit should have close to a non-changing CPU load during the navigation since the incoming topics have fixed rates, fed to the algorithms for perception and state estimation, that should not change. Still, when the Python-based diagnostic tool was subscribed to the topics during the pallet-picking tasks, it increased the CPU load significantly in some cases, even though the handling of the data was minimal. This led to suspicion of high overhead in the RCLPY implementation of the ROS2.

During the ROS2 test, PCs were communicating according to figure 6.8, which shows that one of the PCs broadcasts one topic while the other two subscribe to it and measure the topic rates, bandwidth, CPU load in total, and per core, RAM usage.

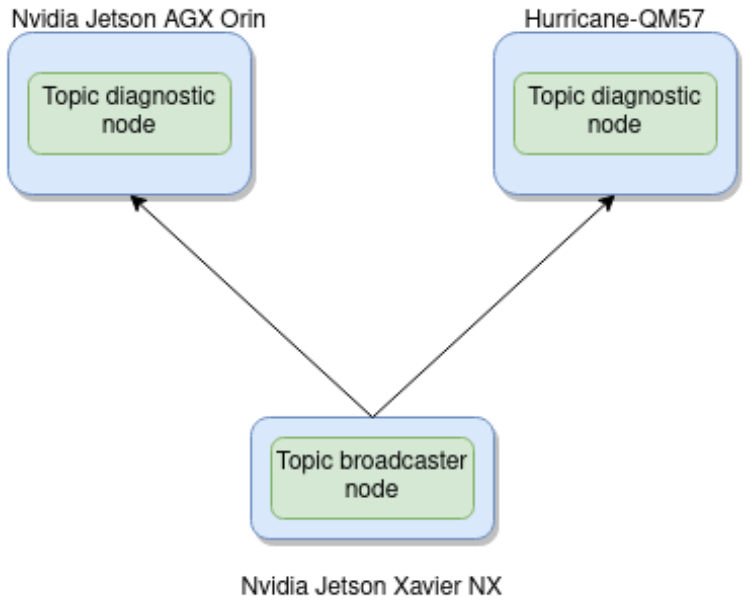

Figure 6.8. *The performance test PC setup for the ROS2 topics*

For evaluation of the ROS client Library for Python (RCLPY), the following tests were conducted. Two different PCs were receiving messages from the third that was sending topics with different payloads. Payloads range from 120 bytes to 100 kilobytes and frequencies from 5 Hz to 350 Hz. The receiving PC specifications can be seen below in the table 6.1.

	Sending PC	PC 1	PC 2
Model	Nvidia Jetson Xavier NX	Hurricane-QM57	Nvidia Jetson AGX Orin
OS	Ubuntu 20.04 LTS	Ubuntu 20.04 LTS	Ubuntu 20.04 LTS
CPU	6-core NVIDIA Carmel Arm®v8.2 64-bit	Intel® Core™ I7 – 610E	Arm® Cortex®-A78AE
CPU MAX FREQ	1.9 GHz	2.53 GHz	2.2 GHz
RAM	16 BG LPDDR4x	2 GB DDR3	32 GB LPDDR5
Ethernet speed	1 Gb/s	1 Gb/s	10 Gb/s
ROS2 version	Galactic	Galactic	Galactic
DDS	Eclipse Cyclone DDS	Eclipse Cyclone DDS	Eclipse Cyclone DDS

Table 6.1. *PC specifications in the test setup*

During the test, the aim is to increase the payload and the transmission frequency to see what happens to the hardware load during the operation and if the payload affects this. This leads to the analysis of the ROS2 subscription rate effect on the actual hardware on two different PCs that are continuously subscribing and measuring the rate and bandwidth of the topics while monitoring the PC's hardware states.

6.2.1 ROS2 performance overhead evaluations results

In the results, there will be a comparison of the resources used by the ROS2 python application while subscribing to different rates of messages with different payloads. There will be a Spearman's linear correlation analysis done on the measured states from the system to see if there is a linear correlation between the resources used in the system and the communication frequency and payload.

Measured results consist of different rates of subscription to different payloads. In figure 6.9, the system rates and CPU load are relatively low payloads, such as 120 bytes and 1000 bytes. The results for the 120 byte broadcasting payload can be seen on the upper side of the figures, and on the bottom is the 1 kilobyte payload.

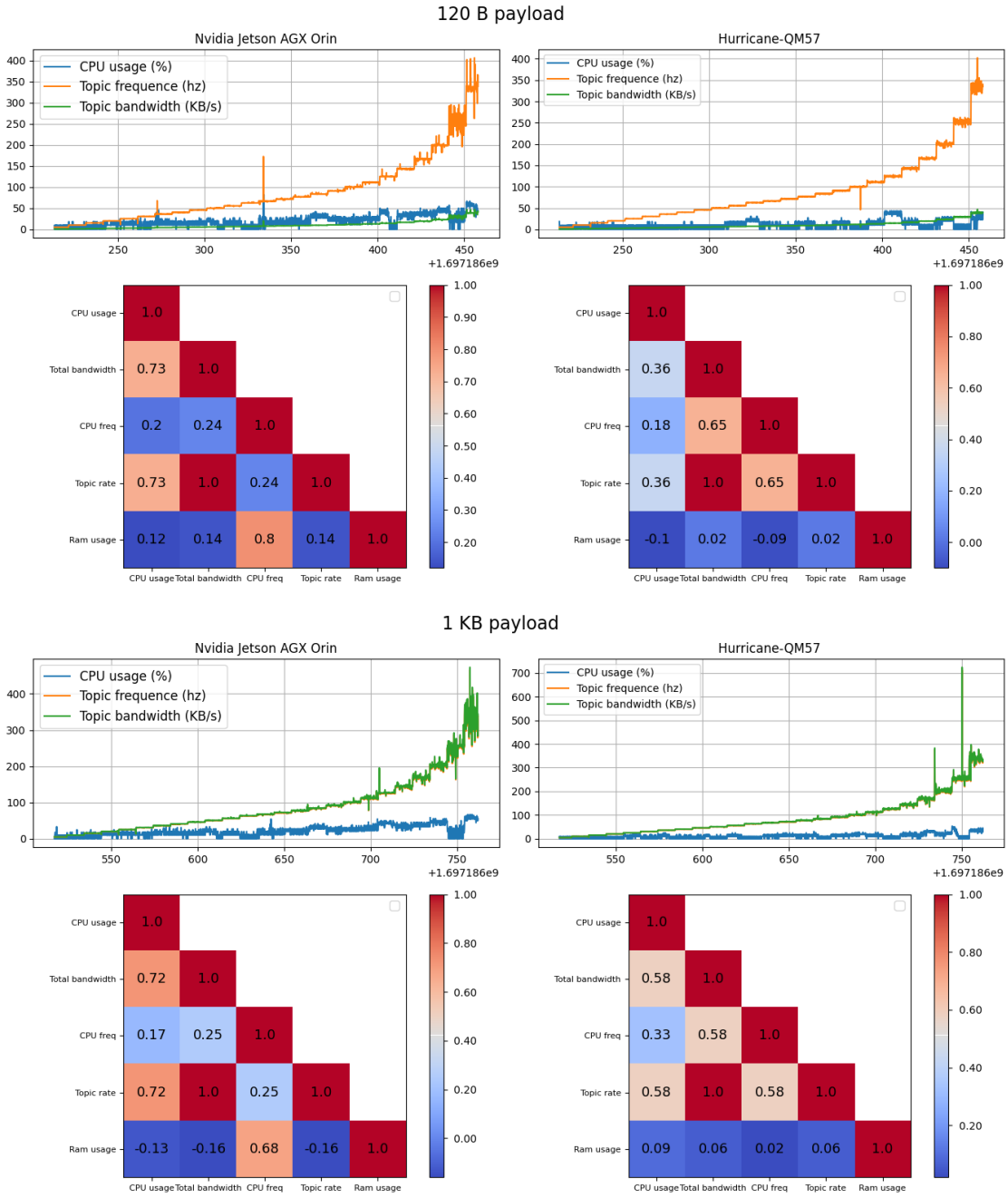

Figure 6.9. *120 B payload and 1 KB payload results to the system*

In these payloads, the system can subscribe to all messages with a given frequency. A notable issue here that can be seen is that the relationship between CPU core usage and the subscription rate is almost linear in the Nvidia Jetson platform. The same linearity does not happen in the Hurricane platform in the lower payload message, but when moving to 1 kB payload, it also starts to increase the stress on the hurricane system, while in the Jetson platform, the correlation stays almost the same.

In the latter tests with higher payload in the figures 6.10, it can be seen that the system fairs well until the CPU load in the Jetson platform reaches maximum utilization, which

causes the receiving system communication rate to aggressively oscillate and be too unreliable for data transmission and subscription. On the other hand, the receiving of the messages completely stops on the other Hurricane platform.

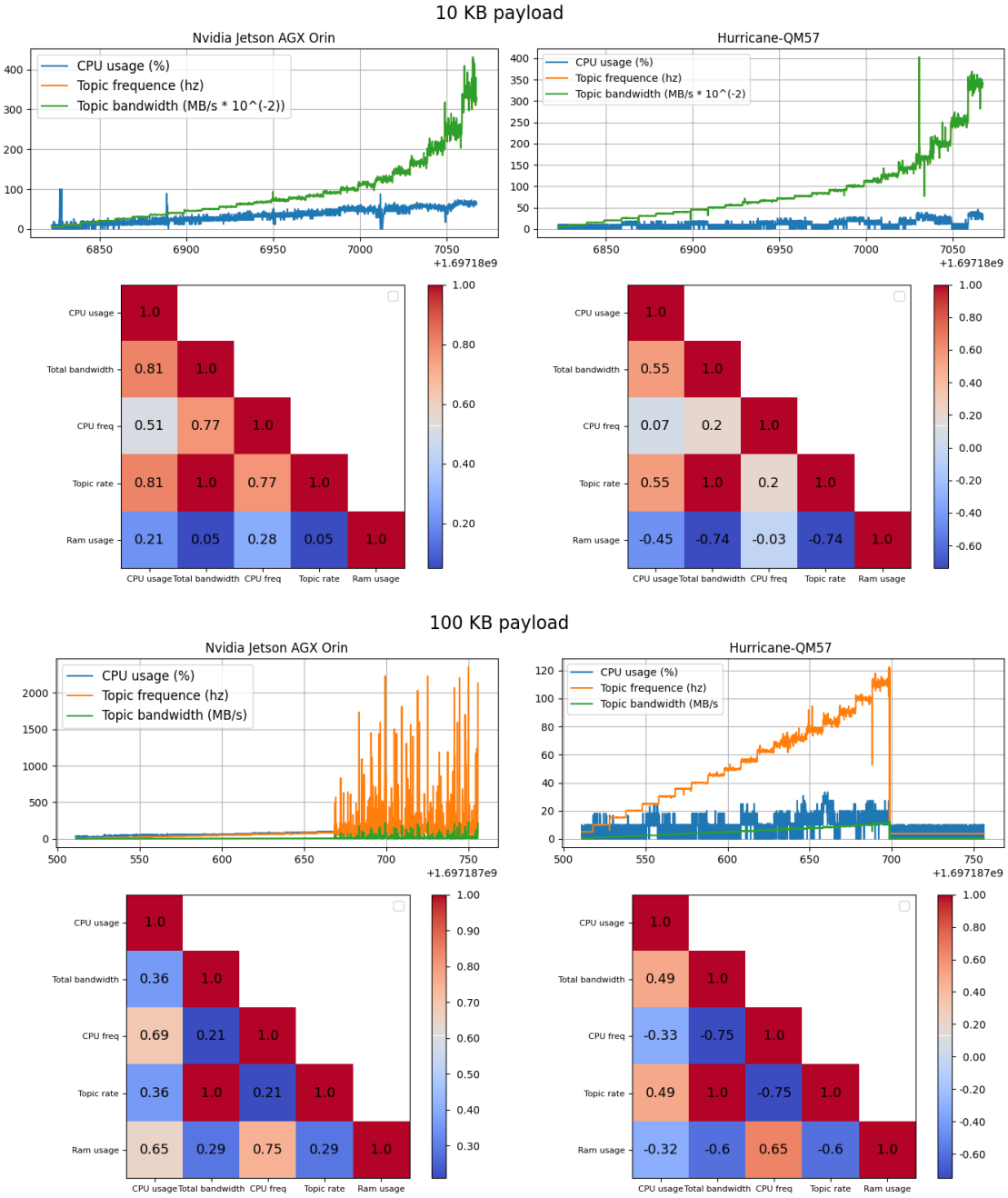

Figure 6.10. *10 KB payload and 100 KB payload results to the system*

The main focal point from the test results is that the subscription rate of the topics has a high linear correlation to the CPU core usage. In the higher rates of the topics, even on the low payload topics, the load gets closer to 30 - 50 % region at only 100 Hz, which is very high usage considering that the data is not deserialized and is discarded immediately after it was received. Very similarly, from the data, it can be seen that when

the system load and the topic frequencies increase, the topic frequency rate becomes more unstable and starts to oscillate around the frequency of the sent topic. On top of the heavy subscription payload, the data usually receives some processing, which also requires computational resources.

Still, given that the load on the system increases as the payload and the frequency increase, the messages can be subscribed to with a certain variance in the rate, but still, they can be received until the payload limit of the system is reached. Still, it is a good sign since it allows communication to flow even in restricted systems. These results could be compared to an implementation with a Real-time kernel to see if the results would vary greatly from the given results here. Though the CPU load would most likely not be affected, it could improve the variance of the subscription rate during the higher frequencies.

Overall system performance with ROS2 is manageable since there are many PCs to divide the resources in the system architecture. From the tests can be seen that the overhead is also dependent on the hardware performance, since the results vary according to the CPU model and performance the overhead also is dependent on this factor in the hardware architecture. Given the high CPU overhead of the RCLPY layer, the high-rate topics should be moved to the RCLCPP (ROS Client Library for C++) layer to avoid the high overhead, especially in resource-limited environments. To avoid the overhead, the module distribution in the system should also be avoided so that the modules implemented to the architecture implement concise functionalities to the system so that one functionality would not be distributed to multiple modules.

7. ANALYSIS OF THE AUTONOMOUS HDMM ARCHITECTURE FOR PALLET-PICKING

Central tools for implementing the architecture and its different layers were behavior trees and ROS2. Combining these layers in the layered architecture implements the system's different design properties or attributes mentioned in section 2.2. In many cases, the various robotics applications are constantly improving, which requires the system architecture to be modular and extendable. This chapter evaluates how the behavior trees and ROS2, in conjunction with the layered architectures, implement these properties and if any of the tools provide significant overhead or design flaws in the base architecture that restrict the future development of the application and architecture. The scope of the evaluation aims to consider how the implemented architecture, with the tools, implements the design properties. The review also considers and discusses if tools like ROS2 and behavior trees offer additional functionalities to support some of the properties that have not been utilized in the use case. Some of these design properties can also be evaluated and realized in the levels of automation. These details will also be reviewed in this chapter. This evaluation includes the related design properties and how they are acknowledged in the levels of automation individually, which are described in the tables 7.1 and 7.2.

Many of the properties and values are evaluated based on the subjective experiences and experiments concluded with the architecture and components during the development of the architecture. In some cases, such as run-time overhead, actual tests were concluded to the ROS2 RCLPY layer 6 to verify the run-time overhead provided to the system. The evaluation includes how the used components support the different design attributes and whether they have internal mechanisms or properties to implement specific design properties with minimal effort.

ROS2, behavior trees, and the layered architecture will be evaluated on a scale between 1 and 5 from the perspective of design properties. Reasoning to scores to the 7.1 will be introduced during the discussion in the text. The previous cell needs to be fulfilled in the table for the system to be on the next level.

Modularity	System components are very time-consuming to change due to multiple functionalities per module	System components are time-consuming to change due to custom dedicated interfaces	Changing system components is possible but requires lot of changes to the connected components	Any singular component can be changed without modifying connected modules	Components and full layers can be changed with any other components or layers implementing the similar functionality
Extendability	Minimal ability to incorporate additional features or functionalities extending the system beyond its initial design	Additional unrelated features can be added to the layers that implement new functionalities	Separate modules can be modified and extended without changing the connected modules	Additional features be added between the existing modules	Capacity for continuous adaptation to easily include new features without affecting the current system in all system layers
Hardware portability	Basic compatibility with specific hardware with minimal adaptability to different platforms and sensors	System supports other basic sensor configurations	System support various platform kinematics	System support varying sensors to support autonomous perception, control, and navigation	The system can operate in any platform with any sensor configuration
Run-time overhead	Substantial increase in processing time or resource demands with critical impact on the system performance	Overhead can be managed, but expanding the system is not possible without optimizing processes or increasing computation capabilities	System overhead is small enough that the system can be extended with additional small features	System can handle additional heavy computational units for additional features	Overhead is not noticeable
Score	**1**	**2**	**3**	**4**	**5**

Table 7.1. *Evaluation matrix of the different design properties on the scale from 1 - 5*

Similarly, the system architecture can be measured in the levels of automation, where certain design properties will contribute to the level of automation in the realized architecture, in a form that it needs to be fulfilled to achieve higher levels of automation.

Reliability	Human monitors the system's capabilities and assesses if the system can continue operation	System can monitor the reliability of its singular state	System can monitor the achievability of the current tasks and report it. The system can recover from erroneous states	System can tell if any task is achievable by the current state in both navigation and manipulation in a static environment	The system can tell if any task is achievable by the current state in both navigation and manipulation in dynamic environment
Robustness	Vulnerable to failures and disruptions with minimal ability to handle unexpected conditions in case of sensor or communication breakdown	Capable of handling some unexpected conditions with limited impact	Capable to handle a range of expected conditions with reasonable effectiveness	Well-equipped to handle a wide variety of unexpected conditions with minimal impact	Designed to withstand and recover from a wide scope of unexpected conditions
Reactivity	System can only react to static and known situations and stop motion if a collision is about to happen	System can react to static and known situations and avoid obstacles	System can perceive and react dynamic and static situations and avoid obstacles	Able to perceive, avoid obstacles, and update the static world model in any structured environments	Able to perceive, avoid obstacles, and update the static world model in any environment
Run-time flexibility	Operation need to be halted to configure the system	Only minor parts of the system can be configured during the run-time	Configuration for all modules can be changed during the run-time	Tasks definition and components can be changed and activated during the run-time	Any component can be changed during the run-time to match the needs of the environment and objective
Autonomy	LoA level 1	LoA level 2	LoA level 3	LoA level 4	LoA level 5

Table 7.2. *Evaluation matrix of the different autonomy properties on the scale from 1 - 5*

Both tables show the scale and definition for each design principle, which leads to a certain score or level of automation in the realized architecture and tools used to realize it. These tables will be gone through from the point of the realized architecture, ROS2, and behavior trees. In some attributes or properties, the given tools do not contribute to the actual realization of the design attribute as much as the underlying implementation that implements the error handling and monitoring. Still under evaluation is to consider whether the tools provide additional features to support the properties.

7.1 Modularity, extensibility, hardware portability, and overhead

System component modularity is an important point in designing the system architecture. To keep the system extensible, the system components should be straightforward to integrate without the need to modify other functionalities around the component. **Modularity** as a design property was referred to point out the system component interchangeability. This property should be regarded when selecting system tools and the architectural de-

sign itself. Modularity and **extensibility** can also be seen as a measure of *coupling* and *cohesion* in the software [55]. The coupling is a measure of independence between the modules in the system architecture [55]. Cohesion measures the degree to which the modules in the system are directed to perform a single task [55].

In the current architecture implementation, many of the system functionalities are individual components that implement a singular purpose inside their designated layers. Communication between the functional modules is implemented through abstract ROS2 communication concepts such as topics, services, and actions. Continuous sources of information such as sensors and localization are implemented over the topics, which comply with the standard ROS2 communication messages like ”Imu”, ”NavSatFix”, and ”Odometry” to provide a stable source of information to components that require them. At the higher level, executing different functionalities is done through the one-time service or long-taking action servers. These interfaces allow the system to have low coupling, which allows changing the system components with relatively low effort without changing implementation in the lower or higher-level modules as long as the new module implements the correct interfaces. Given the need to replace the robot localization, this could be possible by removing the original package and replacing this one with a package that complies with the same sensor interface and provides the same odometry output. Cohesion, on the other hand, is implemented through the tightly coupled modules that implement singular functionalities in the architecture. If for example, the path follower would implement the localization internally, it would make extending and replacing the module much harder in the long run. In this case, the input of one module is most often the output of the other this is considered sequential cohesion, which makes the system cohesion on the higher end. These properties and design choices allow the system to be modular and extensible in all of the layers of the system. For the implementation and the description of the scale, the current state of the architecture implementation falls into the level of five in modularity and extensibility due to any singular component and layer being able to be replaced and extended, given that a new layer or component can be utilized the same communication interfaces. This is not always the case, but due to the abstract interfaces of ROS2, it isn’t too hard to make minor adaptation layers between.

ROS2 and behavior trees also support the modularity and extensibility design attributes at a satisfactory level. As stated before, ROS2 abstract communication methods offer flexibility in the extensibility of the system and allow the implementation of additional functionalities between the modules. Behavior trees, then again, offer flexibility and modularity at the decision-making level. This refers to the system’s ability to implement different behaviors and decision-making in the task execution. Due to the minimal interfacing between the behavior tree modules, the system tree structure can be redefined without needing to change the modules’ implementation. Design in the behavior tree implementations also follows the low coupling and high cohesion design principles. Behavior trees and ROS2

support integration together to provide the ROS2 middleware concepts with the behavior tree action nodes [32][56]. Due to this integration, the behavior tree layer of the application can exist as a separate layer that does not implement any functionalities to the control architecture, but rather interfaces and coordinates the execution of the functional and executive layers. From the tool's point of view, both BT and ROS2 support modularity to an exceptional level, due to the abstract communication, low coupling, and high cohesion in the modules. In behavior trees, modularity is experienced as flexibility in the behavior structures that can be reorganized with low effort, without the need to modify the additional behaviors. This score naturally assumes a design that does not couple external modules together with hard-to-replace interfaces, like internal state representation between the behaviors in the behavior tree. Same with the ROS2 the abstract interfaces allow the system to consist naturally of loosely coupled interfaces that implement concise application layers. For both, the grade on the scale will be five.

Hardware portability relates to the effort to port the system from one platform to another, be it a robotics platform or PC. This design property handles a substantial portion of the architecture. For example, in the robotics platform, the lower level of the functional layer is harder to generalize to the other robotics platforms than the type that it is designed to. In this case, the hardware components are harder to export to similar HDMM systems as seen in figure 5.1, but the higher level components can be adapted more easily as long as the system adaptation layer to the ROS2 (ROS2 CAN interface in figure 5.10) complies with the general ROS2 interfaces to provide the feedback and to control the system. This way, the portable platform does not need the same models of sensors and actuators, just the same abstraction of the sensors to the communication middleware and accurate sensor-to-sensor calibration to provide accurate state estimation, localization, and perception. In the case of ROS2, the abstraction of the common sensors has been implemented in the communication interfaces, which means that many of the general sensors, such as IMUs, lidars, and GNSS antennas, can be integrated into the system effectively. In the other direction, as long as the low-level control system can operate under the assumption that the abstract communication input command to the platform is twist message, which in the platform is turned to the specified control input to the actuators, the architecture can perform in many different platforms. Additionally, in the manipulator, joint velocity is used as an input to the system, which requires some adaptation to the low-level control. In the layered architecture case in the thesis, the system can operate with a similar sensor setup as long as it complies with the ROS2 interfaces in various platforms, in some cases with slight adaptation needed. The main sensor source for perception and world model gathering is lidar or point clouds, which might not be available on all platforms for the perception source, so the system has not yet reached level four portability.

At the behavior tree level, the system abstraction is so high that the behavior tree does

not control the machine but coordinates it. Given that the functional and executive layers are compatible with the robot platform, the decision layer should be able to coordinate the movements of the type of machine. In the higher level, even though the task abstraction is higher, the service and action interfaces are usually unique in some way, which usually requires some adaptation to the communication interfaces. Overall the behavior tree would get five for the hardware portability due to the ability to be platform-agnostic. ROS2, then again, is also a platform-agnostic system, given that the low-level system complies with the interfaces, is not restricted to any type of platform type, and supports the integration of various sensors and drivers to the ecosystem, which scores it level four, due to some adaptation usually required for the hardware layer. Modularity was given a high score because the effort of integrating intermediate adaptability layers is usually low; in this case, the system is evaluated based on pure portability.

One factor that affects or is affected by the many other design properties is the **run-time overhead**. Given the high overhead of the system architecture, it is also hard to extend even if the system architecture would allow it. This means that if the architecture, components, or tools used in the application provide high overhead, the system cannot be extended without refactoring the system components. The chapter 7 conducted tests to measure the overhead that the RCLPY layer in ROS2 exerts on the base system. There was a notice that given the high topic rates and higher bandwidth, the system load increased almost linearly with the topic rates.

In chapter 7, system hardware overhead tests were conducted regarding the ROS2 middleware communication overhead exerted on the distributed system PCs. This chapter will provide some oversights on the actual run time overhead in the application. This analysis focuses on the hardware overhead instead of the communication delay. In the overhead test, it was concluded that in the RCLPY layer of the ROS2, the CPU overhead scaled almost linearly with the topic rate at the communication was sent. This also showed that the payload of the topic has some part in the load, but the main property affecting the overhead is the communication rate. This could lead to a potential system failure during navigation in many low-resource systems. In this case, the internal topic rates can be tuned and are highly up to the implementation. There is still some room to be extended due to the distributed system and implementation system, which allows the overhead compared to the resources to be on level three. This value could be higher, but the ROS2 overhead is quite high overall, which scores the ROS2 overhead also to three. This value is based on the computational load usage overall in the target systems PCs in the table 6.1. Behavior trees did not exert any additional load on the system during the testing, mainly because most of the tree structure is inactive and waits for interfacing with the base system. This is why the behavior trees get a value of five in the overhead.

7.2 Reliability, robustness, reactivity, and run-time flexibility

Reliability, run-time flexibility, reactivity, and robustness highly depend on the underlying implementation of the individual component and how these can handle faulty data and recover from erroneous states. These properties also contribute to the level of automation together, since they cover many aspects of the requirements for the automation level in the 2.1. Reliability and robustness refer to the system's capabilities to handle erroneous data, state estimation, and tasks. Reactivity and run-time flexibility then again measure how the system can react to new situations and changes in the environment and how it can adapt to these changes. In this case, these parameters are evaluated on how the current architecture implements these functionalities and if there are points that could be improved.

The system's ability to detect if the system can perform the given task is called **reliability**. This includes detecting if the plan is not feasible due to certain restrictions such as time constraints or the accuracy of functionalities. At the current state of the architecture, the state and task monitoring mostly rely upon human monitoring, where the task state's accuracy and localization are the monitoring subjects. To improve system reliability there would be a need to add mechanisms that can monitor the certainty of the different states of the machine, task achievability, and additionally recover from them. For example, at the moment of development, the robot localization package is the main localization source for the system. If the updated sensor source GNSS unit does not provide accurate localization data the system will become unreliable very fast, this performance degradation can be detected due to continuous degradation of the filter accuracy, in some cases suitable solution could be changing the localization technique to lidar based if the GNSS connection is not proper. The same goes for the path following and trajectory tracking accuracies, given a certain amount of deviation from the plans it could detect the deviation and replan, at the moment, this isn't the case. Although these examples are specific to some system components, they refer to all of the system components, and they should be able to monitor their functionality and accuracy to some level. Due to the limited reliability of the system, it scores one in the current architecture on the reliability scale. Recognizing the task object, in this case, pallet, could fulfill some of the requirements for task reliability since it can detect the incorrect state and recover from it, but this only applies to a very simple extent compared to what a highly reliable system should be able to detect.

Even though the architecture in question does not yet utilize means to recover or monitor the states and tasks, ROS2 and BT both utilize some functionalities to support reliability. In the behavior tree, there could be an additional monitoring node or the monitoring could be tied to the actions service feedback to monitor the states. Given the degradation of the state certainty, aborting the task would be done, and fallback could try to recover from the state. This recovery could be something of turning on additional recovery or calibration

functions to recover from the state or replan the tasks. The level of reliability depends on the implementation itself, but together the utilized tools of ROS2 and behavior trees can support reliability at higher levels in even dynamic environments. Meaning that the current system could be implemented to provide reliable state monitoring and recovery given extensions to the different layers to provide the feedback and recovery modes.

Robustness refers to the system's ability to handle imperfect inputs and navigate through unknown situations, including communication errors and faulty or incorrect sensor data. Given the numerous abstraction layers within the system, there are multiple points where faulty data could accidentally be injected. Although the ROS2 abstract global communication enables the fast integration of the components, it also allows very accessible communication interfaces. This could potentially lead to other ROS2 interfaces injecting the data into the system by accident if the networks are connected. For example, if two machines are operating in the same network, they could inject the localization data into each other. Unfortunately, the system lacks mechanisms to effectively respond to erroneous sensor data and identify and handle anomalies or external data injections. These erroneous data sources need to be handled somewhere at the driver level to recognize the faulty actuators or sensors. ROS2 provides features to enhance robustness against communication faults through the quality of service settings to address the issues in the lossy communication errors. These settings enable the configuration of communication parameters, allowing for actions like resending or discarding data in network losses, particularly in configurations with lossy networks. In this case, the architecture can employ the quality of service settings to maintain stable communication at different levels, but it cannot employ any measure to check noisy and unreliable sensor data. This is why the robustness is two. As said before the ROS2 employs the QoS settings for more robust communication, but otherwise the robust processing of faulty data mostly needs to be handled in the implementation of the components themselves, this applies in the behavior trees themselves, which only interface with the low-level system. It should not feed faulty data to the system.

Run-time flexibility referred to the capabilities of the system to be configured during the system run-time. ROS2 employs a parameter server, which keeps track of programmed parameters that can be updated during the run-time to configure algorithms and modules. The parameter server makes the modules utilizing ROS2 easily configurable by the user or application layers, such as the decision layer. Similarly, the decision layer should employ run-time flexibility to be able to execute different tasks. As mentioned in the 3.3.1, the behavior tree logic is built in the run-time, allowing changing the mission description at any time, given that all of the required behavior tree nodes are registered to the behavior tree factory at the moment of behavior tree construction. *Generalization* is implemented in the system through the decision layer and behavior trees. Since the system is run-time flexible, it can be configured to implement different task executions that pallet-picking by

defining new behavior trees that utilize the functional layer definitions. If the tasks to be defined are very complicated or large, designing the behavior tree can be hard with the system's general behavior. This may require more concentrated behaviors in the system to define complex actions, like visual servoing.

Reactivity measures how well the system can perceive and react to unknown situations in the environment. This refers to the system's capabilities to detect static and dynamic obstacles while maintaining a stable system state that can avoid these obstacles. In this architecture case, the system can avoid known static obstacles that belong to the known world model. While the nav2 stack does offer features for dynamic obstacle detection during the navigation, that could be utilized, it is not employed in this case to not mix the target pallet as an obstacle. In the current architecture implementation, the architecture sets the reactivity to level two.

7.3 Summary of the evaluation

The evaluation led to a discussion of the properties of the architecture, ROS2, and behavior trees. Figure 7.1 shows the results of the design property analysis.

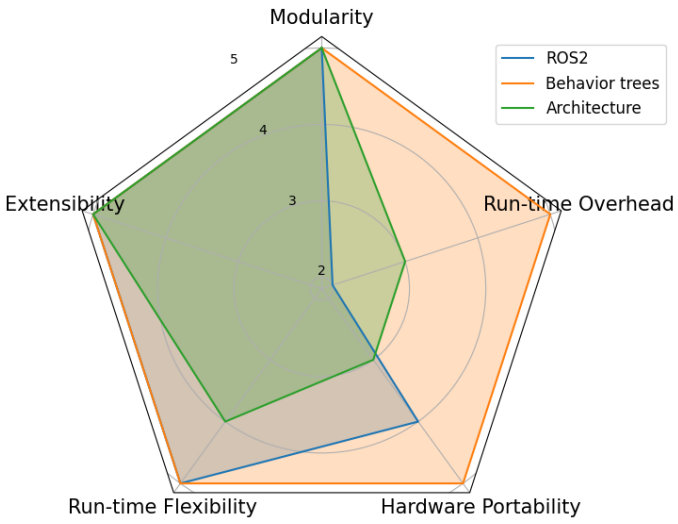

Figure 7.1. *Evaluation results of the design properties*

and 7.1 the architecture autonomy results.

Figure 7.2. *Evaluation results of the autonomy properties of the current architecture*

The analysis includes evaluating each property individually in the system components used for ROS2, behavior trees, and the realized architecture from a scale of one to five according to how the components fill the property description and according to the experiences and experiments with the system components and architecture.

As evaluated in the previous sections, the system architecture with ROS2 and behavior trees offers high flexibility in modularity and extensibility due to the low coupling and high cohesion between the system components and behaviors in the architecture. These properties are important because the target machine is used as a research platform that can experience many changes in the course of the different use cases. With the high modularity, in this case, also comes the trade-off of high overhead provided by the middleware communication. This did not hinder the application but might restrict the future extensibility of the application to even more demanding applications.

From reliability, reactivity, and robustness scale, the system still has room to improve the autonomous capabilities in the architecture to achieve higher autonomy levels. To achieve the higher levels of automation the system still requires improvements to the reliability and robustness of the system to recognize the possible problems in the system to coordinate the system better. The coordination layer or decision layer could still improve the autonomy given the proper feedback from the base system to coordinate the system. These problems relate to the implementation of the independent layers and system components, which should be extended to support more reliable and robust navigation and control for decision-making. ROS2 supports some feedback, communication, and estimation reliability and robustness mechanisms that could be employed to achieve better scale on these properties. Behavior trees, then again, could employ some fallback nodes in the

control nodes on the decision-making level that could verify the system's health or restore the state with additional functionalities.

Autonomy refers to the system's ability to execute tasks without human intervention. In the current state, architecture can perform simple pallet-picking tasks and do simple decision-making to facilitate the task into movements of the machine, which could be achieved with the behavior trees, in this case, as a coordination layer of the system. This architecture is also able to detect if the given initial parameters for the pallet are incorrect. In the levels of automation, this system would be regarded to be in the level between 2 and 3 in both navigation and manipulation tasks. Currently, the system does not fully fulfill the requirements for higher levels of automation, which can also be seen in the previously evaluated properties, which require extensions to higher levels to achieve truly autonomous operations for the target machine. The system can perform autonomous decision-making and perceive the pallet in the world without intermediate authorization from the operator during the task execution. In its current state, the system lacks the higher-level perception capabilities and state estimation that can be recovered in faulty states. Due to the extensibility and modularity of the system, these system components are possible to be added to the system to add obstacle detection and avoidance and other functionalities. Similarly, if the system is not able to converge back to the path, it should be able to generate a new path, but at the moment, it does not do so. In these situations, human intervention is required, which hinders some of the capabilities of the system. Reliability and robustness also affect this property to some degree. If the system's reliability is questionable under certain restrictions or there is faulty data in the system, it will also hinder the system's autonomy due to the system not being able to resolve this autonomously or report this and stop the task execution. From this can be learned that to achieve a higher level of autonomy there first needs to be careful consideration of the other design properties such as robustness and reliability.

8. CONCLUSION

This thesis aimed to recognize the general navigation and control paradigm used in robotics applications and implement one that could utilize mission-level decision-making to accomplish simple everyday tasks in HDMM use cases. The task to be done was pallet-picking, which needed to fulfill many system requirements and features. In the end, behavior trees were a suitable tool to implement mission-level decision-making in the layered architecture, and the architecture managed to fulfill the set system requirements. In conjunction with the behavior trees, this architecture description and implementation could be implemented with the base system ROS2 application stack that existed in the base target system.

From the literature, common architectures for navigation and control could be found, but the hybrid was a very frequent architecture for machines that have to interface with partly observed environments, with support to the coordination layer of the deliberative and reactive control layers. There have been many versions of how to design an architecture that can act fast against local obstacles and other issues while being able to generate obstacle-free paths globally. One of the recognized architectures for hybrid architecture was a layered architecture system. Behavior trees were found to be one of the ways to implement mission-level decision-making for the coordination layer in the layered architecture. Layered architecture offered the idea of setting skills and abstract tasks in the form of compound actions for the machine, making it easier to decompose the actions of the machine for the target application and interface with the decision layer.

Generating high-level abstract tasks such as "generate a path", "follow path", and "move manipulator" offer high-level interfaces in the functional layer that could be easily integrated with other target applications, given that the application can be divided into the skills of the machine able to perform. This skill decomposition also goes hand-in-hand with behavior trees that require the decomposition of the tasks for symbolic-level mission description or to compound actions of the machine. Also, it is possible to change the system layers by following the layered architecture scheme, where the layers are combined with specific middleware layer API definitions. A system can have different decision and executive layers depending on the needs of the use case. These system possibilities also increase the system modularity and flexibility with the given tasks and possible other use cases. Changing components and task definitions becomes easier, allowing

the possibility of extending the base system modules without large modifications to the connected module implementations. Behavior trees were interchangeable to implement different subtrees for different purposes, such as picking and moving the pallet due to the layered architecture formation, where the behavior tree interfaced only with the base system instead of implementing the low-level commanding interfaces. This formulation increased the modularity and flexibility of the system since it allowed for less interaction changeability with the real machine and simulation environment. Similarly, even if the system architecture was generalized, so could the machine interfaces. This leads to fast integration from the simulated machine and environment to the actual hardware due to the abstracted communication interfaces.

Offering multiple different platforms and algorithms, ROS2 can hasten the development process of the heavy-duty mobile machine. On top of the middleware concepts and interfaces, these packages offer different solutions to different applications in the form of localization, state estimation, planning, and control. Robot localization offers flexible localization that accepts multiple different sensors to integrate for state estimation with different Kalman filter approaches. TF2 keeps track of the frame coordinate transformations in the system and could be queried the transformation between any of the connected frames in the transformation tree. Navigation2 and moveit2 concentrate on implementing planning and control in their respective navigation and manipulator task fields. All of these and many other open-source packages contribute to the fast deployment of autonomous heavy-duty mobile machines while allowing the concentrated development of the machine's use case. Overall, ROS2 offers and implements multiple functionalities that are required in many robotics applications that can be utilized to hasten the development of the platform to be autonomous.

In the use case, the pallet-picking application was successfully developed using behavior trees as a component for the decision layer. The application is not without limitations and restrictions. In the tree, the main idea was to minimize the need for replanning since this act can be very demanding from a computational point of view. Since the need for replanning was avoided by utilizing the frame representation of the location of the machine and path being represented in the pallet frame, we could minimize the error to the path by moving the pallet in the global map frame, which allowed the path follower to minimize the error back to the path instead of replanning. The path planner can converge back to the path given that the initial estimation was close enough to the real pose of the pallet and the pallet was detected far enough with the available sensor. Secondly, the system depends on the accuracy of the state estimation and the trajectory and path following the machine, which cannot be affected by the BT during the run-time at the current time, which relates to the system's reliability. Reliability, robustness, and autonomy overall were the points that still require improvement at the decision and architectural level to achieve autonomy without human intervention. There could be an option for the decision layer to monitor

and detect certain reductions in the accuracy of the state estimation, which could lead to a stop, change of state estimation, or human intervention. These are issues of their own, but without accurate sensor calibration and state estimation, the act of pallet picking is very challenging given the accuracy required to navigate the system manipulator forks to the pockets of the pallet, which is one of the key points to have for accurate pallet-picking.

ROS2, as a middleware implementation in the system, offered a lot of flexibility in the development of the system, and middleware offers quite fast integration possibilities for the target system. However, there are still some issues with the Python implementation of the ROS Client Library, such as the high run-time overhead that the base executor implementation brings. This can be seen when the Python nodes handle large frequency topics for varying payloads. The overhead on the CPU core follows almost linearly the topic frequency, which can cause issues on the low-powered devices handling large frequency data like IMUs (15 $Hz \sim$ 500 Hz) or estimating the state of the mobile or manipulator machine. Otherwise, the communication between the system components and the layers was successful enough to enable pallet-picking in the non-real-time operating system. Although the CPU overhead specifically was high in these tests, results differ based on the actual hardware used due to the actual performance of the hardware, such as CPUs per core performance. In this case, the hardware systems used in the actual HDMM show poor software scalability with ROS2, given the high overhead with the RCLPY implementation. In any case, ROS2 was a proper tool for topic interfacing between the system components and implementing the layered architecture communication and system division. Due to the peer-to-peer communication of the ROS2, the distributed control and communication with minimal effort was possible to implement in the system to avoid the CPU overhead in the singular computational unit. The system was able to pick the pallet with the given architecture while maintaining the acceptable load in the actual hardware, although the system was able to implement all of the functionalities due to the distributed load with the given architecture.

Future research could include many things from this field since the act of combining navigation and control with architecture is a wide topic that contains research possibilities. From this thesis topics point of view, relevant research questions could be considered to improve the system and task definition of the machine. There has been some research on generating behavior tree models using machine learning to decompose the high-level task definition to the behavior tree nodes. Could this method be used to generate many types of use cases without programming the behavior tree by hand? For example, large language models are used to automate the design of behavior trees by giving instructions on the task and by providing the API of the behavior trees, which could override the manual labor of designing the manual labor of behavior trees. This could increase the application's design process based on the task's requirements. Additionally, at this stage, the system's reliability, autonomy, and robustness could be increased with addi-

tional functionalities to verify the system's state and control. What would be a suitable metric to evaluate the system state, and what could be done in the state of erroneous state estimation? When should additional path generation be done in the system?

REFERENCES

[1] Machado, T., Ahonen, A. and Ghabcheloo, R. Towards a Standard Taxonomy for Levels of Automation in Heavy-Duty Mobile Machinery. American Society of Mechanical Engineers, 2021.

[2] Barrera, A. *Advances in robot navigation*. eng. Rijeka, Croatia: IntechOpen, 2011. ISBN: 953-51-5539-3.

[3] Macenski, S., Foote, T., Gerkey, B., Lalancette, C. and Woodall, W. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics* 7 (66 2022), p. 6074. ISSN: 24709476. DOI: 10.1126/SCIROBOTICS.ABM6074/ASSET/F1E0C116-9DF3-4C33-A5C1-001E136AFD23/ASSETS/IMAGES/LARGE/SCIROBOTICS.ABM6074-F5.JPG. URL: https://www-science-org.libproxy.tuni.fi/doi/10.1126/scirobotics.abm6074.

[4] Moore, T. and Stouch, D. A Generalized Extended Kalman Filter Implementation for the Robot Operating System. *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer, 2014.

[5] Macenski, S. e. a. The Marathon 2: A Navigation System. *International Conference on Intelligent Robots and Systems (IROS)* (2020).

[6] Coleman, D., Sucan, I., Chitta, S. and Correll, N. Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study. eng. *arXiv.org* (2014). ISSN: 2331-8422.

[7] *tf: The transform library*. Conference on Technologies for Practical Robot Applications (TePRA)., 2013.

[8] Brocke, J. vom, Hevner, A. and Maedche, A. Introduction to Design Science Research. eng. *Design Science Research. Cases*. Progress in IS. Switzerland: Springer International Publishing AG, 2020, pp. 1–13. ISBN: 9783030467807.

[9] Vagia, M., Transeth, A. A. and Fjerdingen, S. A. A literature review on the levels of automation during the years. What are the different taxonomies that have been proposed?: *Applied Ergonomics* 53 (2016), pp. 190–202. ISSN: 0003-6870. DOI: 10.1016/J.APERGO.2015.09.013.

[10] Fassbender, D. and Minav, T. An Algorithm for the Broad Evaluation of Potential Matches between Actuator Concepts and Heavy-Duty Mobile Applications. eng. *Actuators* 10.6 (2021), pp. 111–. ISSN: 2076-0825.

[11] Machado, T., Fassbender, D., Taheri, R., Eriksson, D., Gupta, H., Molaei, A., Forte, P., Rai, P., Ghabcheloo, R., Mäkinen, S., Lilienthal, A. J., Andreasson, H. and

Geimer, M. Autonomous Heavy-Duty Mobile Machinery : A Multidisciplinary Collaborative Challenge. eng. Tampere University. IEEE, 2021.

[12] *Society of Automotive Engineers. "Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles"*. Standard. https://www.sae.org/stand SAE international, 2021.

[13] Mtshali, M. and Engelbrecht, A. Robotic Architectures. eng. *Defense science journal* 60.1 (2010), pp. 15–22. ISSN: 0011-748X.

[14] Reda, A. and Vásárhelyi, J. Model-Based Control Strategy for Autonomous Vehicle Path Tracking Task. eng. *Acta Universitatis Sapientiae. Electrical and Mechanical Engineering* 12.1 (2020), pp. 35–45. ISSN: 2066-8910.

[15] Pendleton, S., Andersen, H., Du, X., Shen, X., Meghjani, M., Eng, Y., Rus, D. and Ang, M. Perception, Planning, Control, and Coordination for Autonomous Vehicles. eng. *Machines (Basel)* 5.1 (2017), pp. 6–. ISSN: 2075-1702.

[16] Siciliano, B. and Khatib, O. *Springer Handbook of Robotics*. eng. 2nd ed. 2016. Springer Handbooks. Cham: Springer International Publishing, 2016. ISBN: 3-319-32552-3.

[17] Jeon, J., Jung, H.-r., Luong, T., Yumbla, F. and Moon, H. Combined task and motion planning system for the service robot using hierarchical action decomposition. eng. *Intelligent service robotics* 15.4 (2022), pp. 487–501. ISSN: 1861-2776.

[18] Palomeras, N., El-Fakdi, A., Carreras, M. and Ridao, P. COLA2: A Control Architecture for AUVs. eng. *IEEE journal of oceanic engineering* 37.4 (2012), pp. 695–716. ISSN: 0364-9059.

[19] Orebäck, A. and Christensen, H. I. Evaluation of Architectures for Mobile Robotics. eng. *Autonomous robots* 14.1 (2003), pp. 33–49. ISSN: 0929-5593.

[20] Ahmad, A. and Babar, M. A. Software architectures for robotic systems: A systematic mapping study. eng. *The Journal of systems and software* 122 (2016), pp. 16–39. ISSN: 0164-1212.

[21] Klančar G. Klancar, G. *Wheeled mobile robotics: from fundamentals towards autonomous systems.* 1st edition. London, England: Butterworth-Heinemann, 2017, 2017.

[22] Bedkowski, J. Mobile Robots. *Mobile Robots - Control Architectures, Bio-Interfacing, Navigation, Multi Robot Motion Planning and Operator Training* (2011). DOI: 10.5772/2304.

[23] Zhu, Y., Zhang, T., Song, J. and Li, X. A new hybrid navigation algorithm for mobile robots in environments with incomplete knowledge. eng. *Knowledge-based systems* 27 (2012), pp. 302–313. ISSN: 0950-7051.

[24] Albore, A., Doose, D., Grand, C., Guiochet, J., Lesire, C. and Manecy, A. Skill-based design of dependable robotic architectures. eng. *Robotics and autonomous systems* 160 (2023), pp. 104318–. ISSN: 0921-8890.

[25] Veres, S. M., Molnar, L., Lincoln, N. K. and Morice, C. P. Autonomous vehicle control systems — a review of decision making. eng. *Proceedings of the Institution of Mechanical Engineers. Part I, Journal of systems and control engineering* 225.2 (2011), pp. 155–195. ISSN: 0959-6518.

[26] López, J., Sánchez-Vilariño, P., Sanz, R. and Paz, E. Implementing Autonomous Driving Behaviors Using a Message Driven Petri Net Framework. eng. *Sensors (Basel, Switzerland)* 20.2 (2020), pp. 449–. ISSN: 1424-8220.

[27] Diab, M., Pomarlan, M., Beßler, D., Akbari, A., Rosell, J., Bateman, J. and Beetz, M. SkillMaN — A skill-based robotic manipulation framework based on perception and reasoning. eng. *Robotics and autonomous systems* 134 (2020), pp. 103653–. ISSN: 0921-8890.

[28] Barnett, W., Cavalcanti, A. and Miyazawa, A. Architectural modelling for robotics: RoboArch and the CorteX example. eng. *Frontiers in robotics and AI* 9 (2022), pp. 991637–991637. ISSN: 2296-9144.

[29] Qureshi, F., Terzopoulos, D. and Gillett, R. The Cognitive Controller: A Hybrid, Deliberative/Reactive Control Architecture for Autonomous Robots. eng. *Innovations in Applied Artificial Intelligence*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1102–1111. ISBN: 9783540220077.

[30] Mohammad, S., Rostami, H., Sangaiah, A. K., Wang, J. and Liu, X. Obstacle avoidance of mobile robots using modified artificial potential field algorithm. (2019). DOI: 10.1186/s13638-019-1396-2. URL: https://doi.org/10.1186/s13638-019-1396-2.

[31] Collendanchise M. Ögren, P. *Behavior Trees in Robotics and AI : An Introduction*. Boca Raton: CRC Press., 2018.

[32] Iovino, M., Scukins, E., Styrud, J., Ögren, P. and Smith, C. A survey of Behavior Trees in robotics and AI. *Robotics and Autonomous Systems* 154 (2022), p. 104096. ISSN: 0921-8890. DOI: 10.1016/J.ROBOT.2022.104096.

[33] Ghzouli, R., Berger, T., Johnsen, E. B., Wasowski, A. and Dragule, S. Behavior Trees and State Machines in Robotics Applications. eng. *IEEE transactions on software engineering* 49.9 (2023), pp. 4243–4267. ISSN: 0098-5589.

[34] Huang, Y. e. a. A Robot Architecture of Hierarchical Finite State Machine for Autonomous Mobile Manipulator. *Intelligent Robotics and Applications.* (2017).

[35] Ben-Ari, M. and Mondada, F. *Elements of Robotics*. eng. 1st ed. 2018. Cham: Springer Nature, 2018. ISBN: 3-319-62533-0.

[36] *MIURA Yasuyuki. Distributions (2023)*. URL: http://wiki.ros.org/Distributions (visited on 12/07/2022).

[37] *Apache Software Foundation, Apache License, Version 2.0*. URL: https://www.apache.org/licenses/LICENSE-2.0.html (visited on 12/18/2023).

[38] *Open Robotics. Understanding nodes (2023)*. URL: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html (visited on 08/07/2022).

[39] Cruz, J. M. e. a. DDS-based middleware for quality-of-service and high-performance networked robotics. *Concurrency and computation.* (2012).

[40] *Open Robotics. Understanding services (2023)*. URL: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html (visited on 08/07/2022).

[41] *Open Robotics. Understanding actions (2022)*. URL: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html (visited on 08/07/2022).

[42] *On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward.* Proceedings of the National Academy of Sciences - PNAS, 2021.

[43] *Why Gazebo?* URL: https://classic.gazebosim.org/ (visited on 12/18/2023).

[44] *Rviz2*. URL: https://turtlebot.github.io/turtlebot4-user-manual/software/rviz.html (visited on 09/20/2023).

[45] Rico, F. M. *A Concise Introduction to Robot Programming with ROS2*. eng. 1st ed. Milton: CRC Press, 2022. ISBN: 9781032267203.

[46] *TurtleBot3*. URL: https://www.turtlebot.com/turtlebot3/ (visited on 01/04/2023).

[47] *Integrating GPS Data*. URL: https://docs.ros.org/en/melodic/api/robot_localization/html/integrating_gps.html (visited on 01/04/2023).

[48] *REP-105*. URL: https://www.ros.org/reps/rep-0105.html (visited on 01/04/2023).

[49] Macenski, S., Martin, F., White, R. and Ginés Clavero, J. The Marathon 2: A Navigation System. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020.

[50] Yossawee, W., Tsubouchi, T., Kurisu, M. and Sarata, S. A semi-optimal path generation scheme for a frame articulated steering-type vehicle. eng. *Advanced robotics* 20.8 (2006), pp. 867–896. ISSN: 0169-1864.

[51] Malvido Fresnillo, P., Vasudevan, S., Mohammed, W. M., Martinez Lastra, J. L. and Perez Garcia, J. A. Extending the motion planning framework—MoveIt with advanced manipulation functions for industrial applications. eng. *Robotics and computer-integrated manufacturing* 83 (2023), pp. 102559–. ISSN: 0736-5845.

[52] *Move Group C++ Interface*. URL: https://moveit.picknik.ai/humble/doc/examples/move_group_interface/move_group_interface_tutorial.html (visited on 01/17/2023).

[53] *Trimble SPS351 GPS Receiver*. URL: https://www.geos.ed.ac.uk/~gisteac/fieldkit/userguides/gps/survey_grade_basestations/trimble/SPS855%20GSG/SPS351%20GSG%20PDF.pdf (visited on 12/18/2023).

[54] Janos, D., Kuras, P. and Ortyl, Ł. Evaluation of low-cost RTK GNSS receiver in motion under demanding conditions. eng. *Measurement : journal of the International Measurement Confederation* 201 (2022), pp. 111647–. ISSN: 0263-2241.

[55] Chowdhury, I. and Zulkernine, M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. eng. *Journal of systems architecture* 57.3 (2011), pp. 294–313. ISSN: 1383-7621.

[56] *Integration with ROS2*. URL: `https://www.behaviortree.dev/docs/ros2_integration` (visited on 01/07/2023).