NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

JOÃO NUNO SALVADOR RODRIGUES CORVO

Bachelor in Electrical and Computer Engineering

BEHAVIOR BASED AUTONOMOUS MOBILE ROBOT FOR INDUSTRIAL LOGISTICS

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September, 2022

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

BEHAVIOR BASED AUTONOMOUS MOBILE ROBOT FOR INDUSTRIAL LOGISTICS

JOÃO NUNO SALVADOR RODRIGUES CORVO

Bachelor in Electrical and Computer Engineering

Adviser: José António Barata Oliveira
Full Professor, NOVA University Lisbon

Co-advisers: Francisco Antero Cardoso Marques
Research Engineer, NOVA University Lisbon

Magno Edgar da Silva Guedes
Senior Software Researcher, Introsys

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September, 2022

Behavior Based Autonomous Mobile Robot for Industrial Logistics

I want to dedicate this dissertation to my parents, grandmother,
and all of my friends who supported me, unconditionally,
during my master's degree. Thank you!

Acknowledgements

First of all, I want to acknowledge the institution of NOVA School of Sciences and Technology and all of its community to have welcomed me as well as they knew and for making me a part of it. In particular, I want to express my gratitude to professors José Barata and Francisco Marques for the opportunity to start my journey, in robotics, working in The Robotics & Industrial Complex Systems Group of UNINOVA as a researcher.

Secondly, I would like to give a word of appreciation to Introsys and all of my colleagues in the RDI department, specifically, to my supervisor, Magno Guedes for giving me the opportunity to apply my work to a real research and impactful project and giving me all the support they could to make it successful.

Last but not least, I want to express my enormous gratitude to my parents, family, and friends who supported me through thick and thin and gave me the strength to never give up.

“"There is nothing impossible to they who will try."”
(Alexander the Great)

Abstract

The design of robot behaviors to meet the requirements of the new industrial era - Industry 4.0 - has grown significantly in recent years. Especially the demand for flexible and adaptable systems has increased exponentially since intelligent robots started to be integrated into assembly lines and replace human activities.

Tools such as Finite State Machines have proven to be an understandable and quick way to solve high-level problems in robotics; however, unmanageable when complexity rises. They become confusing and unreadable, making their modification and maintenance a problem. New tools, such as Behavior Trees, have emerged, creating modular, flexible, and adaptable systems without sacrificing readability with the increased complexity.

The proposed architecture follows a hierarchical layered approach taking advantage of Behavior Trees, developing modular robot skills and system interfaces to create an autonomous behavior-based system. The software was implemented and tested in an Autonomous Mobile Robot capable of navigating complex environments and executing basic tasks.

The results showed real advantages in using the layer-based approach, particularly giving the system modularity and increased flexibility capable of being easily improved and used in other systems. It was also concluded that Behavior Trees are an adequate tool for reactive systems in highly dynamic environments.

Keywords: autonomous mobile robots, behavior trees, industry 4.0, autonomous navigation, path planning, ROS, layer-based approach, robot skills

Resumo

Nos últimos anos, tem-se verificado um crescimentos na modelação de comportamentos robóticos com o objetivo de satisfazer necessidades dos novos paradigmas da indústria. Em particular, na indústria 4.0, com a integração de robôs nas linhas de produção e a substituição dos humanos em diversas atividades, tem-se verificado um aumento na exigência de sistemas mais adaptáveis e flexíveis.

Ferramentas tais como as máquinas de estado provaram ser percetíveis e de fácil utilização na resolução de problemas na área da robótica. No entanto, com o aumento da complexidade, tornam-se problemáticas pela sua desorganização e ilegibilidade. Por conseguinte, emergiram novas estruturas, tais como as árvores de comportamento, capazes de tornar os sistemas mais modulares e flexíveis.

A arquitetura por hierarquisação de camadas proposta, tira partido das vantagens das árvores de comportamento, com o desenvolvimento de comportamentos e interfaces de modo a criar um sistema reativo e autónomo. O software foi implementado e testado num robô móvel autónomo, capaz de navegar em ambientes complexos e de executar tarefas basicas.

Os resultados mostraram vantagens na utilização da arquitetura proposta, em particular, trazendo modularidade e flexibilidade ao sistema robótico, permitindo uma futura melhoria de cada um dos módulos, tal como, a sua utilização noutros sistemas.

Palavras-chave: robôs móveis autónomos, árvores de comportamento, indústria 4.0, navegação autónoma, planeamento de rotas, ROS, abordagem por camadas, abilidades robóticas.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AGV Autonomous Guided Vehicle 3

AI Artificial Intelligence 1, 23

AMCL Adaptive Monte Carlo Localization 15

AMR Autonomous Mobile Robot xix, xx, xxi, 2, 3, 4, 5, 7, 36, 42, 47, 55, 56, 57, 58, 59, 60, 61, 63, 64, 65, 66, 67, 68, 69, 71, 72, 73, 75, 80, 99, 102, 103, 104, 105, 106, 107

ARM Advanced RISC Machine 58

BT Behavior Tree xix, xx, xxi, xxiii, xxv, 2, 3, 4, 5, 7, 17, 19, 20, 21, 22, 23, 24, 25, 35, 36, 37, 38, 39, 40, 42, 43, 44, 46, 48, 49, 51, 55, 56, 60, 61, 62, 63, 65, 67, 69, 70, 72, 73, 74, 75, 76, 77, 79, 80, 81, 91, 92, 93, 94, 95, 101

CONTIGO Robótica **Co**laborativa I**n**teligente para O**ti**mização Er**go**nómica de Processos Industriais 2, 4, 47, 48, 66, 80

CPU Central Processing Unit 57, 58, 60

DA Data Analytics 1

DWA Dynamic Window Approach xix, 12, 13, 14, 15, 16

EB Elastic Band xix, 13, 14, 15, 16

eMMC Embedded Multimedia Card 58

FSM Finite State Machines xix, 2, 16, 17, 20, 21, 22

GB Gigabytes 57, 58, 60

GPU Graphics Processing Unit 57, 58

GUI Graphical User Interface 24, 25

hp Horsepower 57
Hz Hertz 60

IoT Internet of Things 1

kW Kilowatt 57

ML Machine Learning 1

RAM Random Access Memory 57, 58, 60
ROS Robot Operating System xix, xxv, 4, 5, 9, 11, 12, 14, 23, 24, 25, 27, 28, 29, 30, 31, 32, 34, 36, 42, 43, 46, 47, 55, 56, 60, 73, 74, 75, 76, 79, 80

SSD Solid State Drive 57, 58, 60

TEB Time Elastic Band xix, 12, 14, 15, 16, 56

VWAE Volkswagen Autoeuropa 2, 4, 47, 80

XML Extensible Markup Language xxv, 24, 25, 35, 51, 56, 60, 62, 72

YAML Ain't Markup Language 40, 69, 75, 80

1

Introduction

1.1 Context and Motivation

The term Industry 4.0 was first introduced by a German Federal Government initiative to strengthen the cooperation between universities and private companies. The main goal was to develop advanced production systems to increase the productivity and efficiency of the national industry [1]. This new industrial stage emerged to change the way manufacturers and production systems work and reframe the human role in such environments.

The intention is to approach the working activities in the production chain smartly and more efficiently (*Smart-Working*), integrating technologies such as Internet of Things (IoT), Machine Learning (ML), Artificial Intelligence (AI) and Data Analytics (DA) to improve information quality from production processes, communication and build improvement areas' awareness.

In Industry 4.0, the systems are meant to automatically adapt to numerous types of products and conditions, increasing production efficiency and reducing the number of resources needed (*Smart Manufacturing*). Therefore, these advanced and flexible systems enable a deeper layer of customized products leading to faster innovation and manufacturers' competitiveness.

Advanced robotics is one of the technology pillars of Industry 4.0 [2]. In fact, accordingly to the Association for Advancing Automation [3], the third quarter of 2021 set the record in robot sales, in North America, with nearly 29 000 units valued at 1.48 billion dollars [4]. In Europe, a growth of 15% in sales corresponding to 78 000 (seventy-eight thousand) was reported by the International Federation of Robotics (IFR) [5]. Particularly, for Automated Guided Vehicles (AGVs) and Autonomous Mobile Robots (AMRs) market, a report predicted an opportunity worth more than 18 billion dollars by 2027 with an installed base of 2.4 million robots [6].

Robots are useful to perform repetitive and heavy-weight tasks such as predictive maintenance, monitoring machines, transport, and handling heavy objects. Although there are simple tasks such as screwing or wielding that don't require complex behaviors

and decision-making processes, there are other situations that robots must have those skills, such as transporting objects in warehouses with human presence, guiding people, or avoidance in dynamic and changing environments. Hence, in those situations, robots must be aware of their surroundings and make decisions. For instance, during the transportation of an object, the robot is confronted with a blocking obstacle in its path. In that situation, identifying the obstacle and, consequently, the environment is mandatory, and a decision to reroute its trajectory and take another collision-free path to the goal is expected.

Therefore, decision-making processes lead to an execution of a new set of actions to achieve a defined goal. Although humans have the capacity to adapt quickly to changes and rapidly establish new plans, robots need to know how to behave in different situations and plan their actions accordingly. Thus, a roadmap of choices has to be designed, in detail, for machines to make decisions, modeling the human thought process in the simplest way possible.

Finite State Machines (FSM) and Behavior Trees (BTs) are the most used design tools to model robots' decision-making skills. They use conditions, actions and other processes to describe multiple behavioral possibilities and accomplish adaptation and flexibility. FSM are widely used not only in robotics but in other fields as well, such as electronics [7], cloud-based networks [8] and cryptocurrency's smart contracts [9]. However, in recent years, Behavior Trees emerged to model decisions and behaviors in computer games, and they easily spread across other fields such as robotics [10], because of their advantages. Among them are the increased modularity, scalability, and reusability without sacrificing readability and complexity.

Moreover, modeling robot behaviors using BTs increases efficiency in the development of new recovery mechanisms for robots. Encapsulating complex tasks in easy structures makes them easy to modify, transform, and even reuse the same actions across different use cases. Thus, the development of architectures benefiting from those advantages must be explored and tested in robots to assure credibility, security, and usability in different scenarios.

This dissertation explores the aforementioned advantages of Behavior Trees with a newly implemented architecture. The proposed model is validated in an Autonomous Mobile Robot (AMR) which belongs to a project called CONTIGO [11], in development by INTROSYS [12] and Volkswagen Autoeuropa's [13] partnership.

1.2 Path for Contribution

In robotics, there are several research projects using BTs to model behaviors. *Rodiva et al.* [14] introduce an improved version of the classical BT, called *extended behavior trees (eBT)*. The extended version not only describe how to execute a behavior but also consider its effects on the world state. Thus, its possible to optimize the actions' execution in order to reduce resources and time spent. Another study conducted by *Giunchiglia et al.* also

extended the classical model of *BT*s and introduced the concept of conditional behavior trees (CBT). These new concept enables, through pre- and post-processing conditions, to verify the executability of BTs. With the different focus, namely BT design automation, there are studies which take advantage of machine learning methods and algorithms such as *reinforcement learning (RL)* [15] and *genetic algorithms* [16] to model and optimize behaviors. Finally, Colledanchise and Natale [17] showed how synchronization between BTs impact processes' efficiency and the performance of systems [18].

Some of them are BT's fundamental theory improvements, others are focused on robot manipulation, and a few approach robot navigation. Thus, it is difficult to find real-world implementations of robotic behaviors, especially in industrial environments. Among the reasons are the rules and safety procedures that robots must follow and possess to work in such environments. Furthermore, some industrial places are highly dynamic, meaning that humans and machines are in movement on the shopfloor. Also, in factories there are reusable workplaces, that is, they are not destined for just one task, but instead, they usually hinge on the product being manufactured.

Although there are a few experiments, such as a self-adaptive task management layer using BT's and reinforcement learning for multiple AGVs, in a manufacturer shopfloor [19] and a human-aware collaborative robot [20], there are numerous topics to research, improve and validate. *Hao Hu et al.* [19] pointed out that validation in a real-world scenario and algorithmic efficiency improvement is required to perceive the real advantages of BTs.

Therefore, this dissertation aims to contribute to the application of Behavior Trees in Autonomous Mobile Robots, demonstrating and validating their advantages using newly researched methods for their implementation.

1.3 Proposed Approach

The work developed is based on the guidelines defined by the RobMoSys Robotic Software Component [21], which encourages the composition of robotics applications following model-driven techniques.

In this work, the robotic software will be categorized in:

- **Task Layer**, which defines how the robot accomplishes a goal. In particular, it defines the Behavior Tree;
- **Skill Layer** where the basic capabilities of the robot are designed, for example, grasp an object or navigate to a given location;
- **Interface Layer** serves as the access point for the skills to communicate with the robot.

The other RobMoSys abstraction layers namely *Mission*, *Function*, *Operating System*, *and Hardware* were not implemented or included in the developed software.

The proposed architecture aims implement certain behaviors using Behavior Trees for an Autonomous Mobile Robot (AMR) running the Robot Operating System (ROS) and validate their advantages. The robot in which the implementation will be tested is in the scope of the project Robótica **Co**laborativa I**n**teligente para O**ti**mização Er**go**nómica de Processos Industriais (CONTIGO) [11] by INTROSYS [12] and Volkswagen Autoeuropa [22].

It proposes an Autonomous Mobile Robot with a robotic arm on the top, capable of transporting car parts in the Volkswagen Autoeuropa shopfloor. Therefore, the AMR must have the ability to navigate securely and autonomously inside the factory, which includes avoiding both static and dynamic obstacles such as containers, forklifts, or people. As a collaborative robot, it must identify operators and be capable of interacting with them, having defined behaviors (routines) in that sense. Moreover, in alignment with the goal of the project, it will also have pick and place routines in order to deliver the necessary car parts to the operators for their work. Another characteristic of the AMR includes voice and gesture recognition to execute certain tasks. The Figure 1.1 shows an illustration of the project's robot prototype.

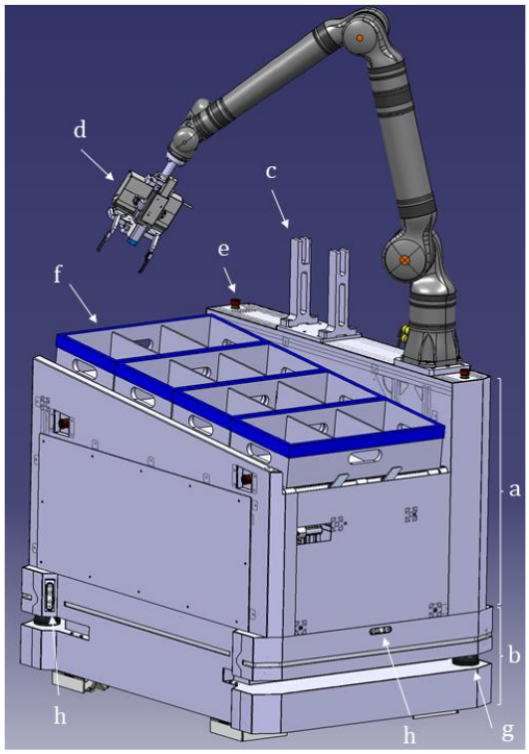


Figure 1.1: 3D view of the AMR's structure. a) Upper structure of the AMR b) Bottom structure of the AMR c) Support structure to revert pieces' orientation; d) Gripper; e) Emergency button; f) Pieces' transport boxes; g) Laser scanner; h) Camera.

In the development of the suggested architecture, the following technologies are used: the BT library *BehaviorTree.CPP* [23], the BT visualization tool, *Groot* and Robot Operating

System (ROS), the main operating system for robot applications development. Also, all functional tests and validations will be performed on Gazebo, the ROS default simulator, and visualized in RVIZ.

Nonetheless, a thankful word and acknowledgment have to be given to INTROSYS for all the availability and support for the development of this dissertation.

1.4 Dissertation Structure

The following chapters of this document present a detailed description of the steps followed in order to develop robot behaviors according to a generic architecture based on abstraction layers. The chapters are structured as follows:

- *Chapter 1: Introduction* presents the basic pillars of Industry 4.0 and some important concepts, describes the contribution of this dissertation for the implementation of behaviors in Autonomous Mobile Robots using Behavior Trees, and exhibits the approach followed.
- *Chapter 2: State of the Art* provides an overview of previous work accomplishments in robot navigation and robot behaviors, as well as a detailed explanation of key concepts. Possible flaws and solutions are explored and summarized.
- *Chapter 3: Supporting Concepts* outlines the most important concepts to understand better the architecture and the mechanisms behind the work developed in this dissertation, as well as the technologies used.
- *Chapter 4: Architecture and Implementation* provides a detailed description of the proposed architecture and its implementation. It also analyses the choices made for specific aspects of the architecture.
- *Chapter 5: Experimental Results and Discussion*. The experimental results of the developed architecture and the approach followed are exposed, discussed and analyzed.
- *Chapter 6: Conclusions and Future Work* summarizes the implementation made and achievements. Comments and improvements are considered. Future work and research are acknowledged so that progress can be made on the topic of this dissertation.

2

State of the Art

The goal of this chapter is to identify the research gap that motivated the work herein presented by surveying the recent contributions of the robotics community to the topics of autonomous navigation and robot motion behavior. The main focus is motion behaviors because there are other possible types of behaviors, such as social or interactive, that are out of the scope of this dissertation.

Section 2.1 explores robot *path planning*, distinctions between groups of planners and algorithms are compared and analyzed, and the Section 2.2 describes *Behavior Trees* as a solution to model robots' decision-making process and principles of robot navigation.

2.1 Path Planning

The main output of motion behaviors is a motion planning. Thus, it is important to first understand the concept of path planning both from global and local perspectives.

In this Section, a brief explanation of concepts such as path planning - global planning and local planning - are performed. Some algorithms which belong to global planners and local planners are explained and analyzed. Finally, the most well-known and widely used planners are compared regarding certain characteristics as performance and movement time.

The core functionality of an AMR is the ability to navigate safely and autonomously in a human-centered environment (not built specifically for unmanned vehicles). Consequently, to achieve an endpoint (goal) from the actual location, the robot must have a navigation system capable of creating a collision-free trajectory, that is, a path, to its final destination. During the navigation, the competence of identifying and avoiding obstacles is paramount in a AMR, in order to perform it safely. Preferably, the planners' calculations try to achieve the shortest trajectory possible based on some principles and algorithms. In robotics, this concept is called path planning.

The path planning process has two components: the *global planning* and the *local planning*. *Global planners* approach the path planning process broadly, as their goal is to compute a path from an initial point to an endpoint. On the other hand, *local planners* are

responsible for traversing the path calculated by *global planners* while dealing with the uncertainties of the environment, the data produced by the sensors, and the robot's kinematics and motion [24]. Both global and local planners use waypoints to navigate, that is, reference points in the middle of the trajectory, to compute it as optimally, smoothly, and shortly as possible without colliding with obstacles [25].

Path planning algorithms and complexity change depending on the environment and the localization precision. For instance, structured environments, distinctive landmarks, and static objects are easier to navigate than dynamic, uneven, and unstructured environments continuously changing. For this reason, sometimes custom modifications must be made to the most common algorithms to accommodate the demands of the environment in question.

In the next sections, global and local planners with their corresponding algorithms will be specified and analyzed.

2.1.1 Global Planners

Global Planners require a *a priori* representation of the environment, commonly a map, to compute the best route possible. Optimal paths and map analysis depend on the algorithm used, and they can be divided into categories. According to Cai *et al.*, [26] there are three categories: graph search-based algorithms, random sampling algorithms, and intelligent bionic algorithms. The most widely-known and frequently used graph search-based algorithms include the Dijkstra algorithm [27] and A* (A-Star) [28]. Regarding random sampling algorithms, based on the research, Rapidly-exploring Random Tree (RRT) [29] and its variants, such as Risk-based Dual-Tree Rapidly exploring Random Tree (Risk-DTRRT), [30] are widely implemented in robotics. Finally, there are algorithms created in order to demonstrate some kind of intelligence, the intelligent bionic algorithms. They were based on insect behaviors - Ant Colony Algorithm (ACO) [31] and Particle Swarm Optimization Algorithm (PSO) [32] - and genetics - Genetic Algorithm (GA) [33].

Graph search-based algorithms are optimal for 2D structured environments where navigation is relatively simple with static objects. However, using them with large maps or high-dimensional environments can cause a computational overload. That is, its computation of global planners increases proportionally with the map size. On the other hand, random sampling algorithms, compared to graph search-based algorithms, are much more efficient and widely adopted in dynamic and multi-dimensional environments. Therefore, algorithm must be chosen correctly depending on the environment to achieve optimal performance.

Furthermore, global planning algorithms all suffer from the *local optima problem*. The problem is that the first optimal path found is the result of the algorithm. This behavior is considered optimal because better solutions can exist, and they are not explored by the algorithm. However, there are proposed solutions based on the fusion of some of

the aforementioned algorithms, such as the Genetic Algorithm-Particle Swarm Optimization Algorithm (OGA-PSO), [34] or adaptation of others imposing some multi-objective constraints to compute the optimal global path [32].

2.1.1.1 Algorithms

Since the Dijkstra algorithm and A* are two widely used and implemented algorithms in ROS supporting the majority of *global planners*, they are explained in detail, in the next sections.

Dijkstra's Algorithm

As a graph search-based algorithm, Dijkstra [27] is used for finding the shortest path between two nodes in a graph. With knowledge of the start and end nodes, the algorithm calculates the path with the lowest distance between the two.

At first, all vertices of the graph, i.e., distances between two linked nodes, are set to infinity, meaning that their actual distance to the goal is unknown. Hence, they were considered unvisited. The source node is marked with zero distance and as a current node. After that, all the distances between the source node to its unvisited neighbors' are computed, considering that the shortest path was not found yet. The shortest distance between them is saved, and the visited node is removed from the list and set as visited. A visited node will never be checked again. In case of the goal node is visited, or all the unvisited set nodes are marked with infinity, the algorithm stops the search, meaning that the shortest path was found. Otherwise, the current node is assigned as the shortest distance node, and the algorithm continues the search. The algorithm 1summarizes the previous explanation.

A* Algorithm

The A* (A-star) algorithm is an extension of the previously explained Dijkstra algorithm. This algorithm is similar to Dijkstra's but also approaches the search problem with a heuristic approach. Although there's a possibility of finding a sub-optimal path instead of the shortest one, the heuristics-oriented search guide proves to achieve better performance. Heuristics define an approximate distance to the goal node, sometimes being shorter than the real distance.

The algorithms' function - $f(n) = g(n) + h(n)$ - is used to calculate the distance between the initial node and the goal node, passing by the n node. The $g(n)$ is the effective distance from the initial node to the node n. The $h(n)$ is defined as the heuristic value estimation of the distance between node n and the goal node. The A* algorithm starts on the initial node and calculates the distance $f(n)$ of its neighbors. After choosing the shortest distance neighbor and the procedure is repeated until the goal node is reached. The following pseudocode (algorithm 2) summarizes the a-star algorithm.

Algorithm 1: Pseudocode of Dijkstra's Algorithm [35]

```
1:  Function Dijkstra(graph[n], source) :
2:  for node in graph do
3:     dist[node] ← inf;
4:     path[node] ← und;
5:  end for
6:  dist[source] ← 0;
7:  Q ← G[N];
8:  while G[N] is not empty do
9:     small_dist_node ← smallestDistNode(G[N]);
10:    remove small_dist_node from G[N];
11:    for each neighbor of small_dist_node do
12:       new_dist ← dist[small_dist_node] + dist_between(small_dist_node, neighbor);
13:       if new_dist < dist[neighbor] then
14:          dist[neighbor] ← new_dist;
15:          path[neighbor] ← small_dist_node;
16:       end if
17:    end for
18: end while
19: return path[];
```

Algorithm 2: Pseudocode of A*(A-Star) Algorithm (adapted from [35])

```
1:  Function Dijkstra(graph[n], source) :
2:  for node in graph do
3:     dist[node] ← inf;
4:     path[node] ← und;
5:  end for
6:  dist[source] ← 0;
7:  Q ← graph[n]];
8:  while graph[n] is not empty do
9:     small_dist_node ← smallestDistNode(G[N]);
10:    remove small_dist_node from graph[n];
11:    for each neighbor of small_dist_node do
12:       new_dist ← dist[small_dist_node] + dist_between(small_dist_node, neighbor);
13:       new_dist_heur ← dist[small_dist_node] + distBetween(small_dist_node,
          neighbor) + heuristicFunc(small_dist_node, neighbor);
14:       if new_dist_heur < dist_heur[neighbor] then
15:          dist_heur[neighbor] ← new_dist_heur;
16:          dist[neighbor] ← new_dist;
17:          path[neighbor] ← small_dist_node;
18:          if 'GOAL' in graph[n] then
19:             break;
20:          end if
21:       end if
22:    end for
23: end while
24: return path[];
```

Discussion

Regarding Dijkstra's algorithm, despite finding the shortest path between two nodes, it is not the most efficient possible due to its unfocused search. This means that, as it doesn't know which direction to search for the goal node, time can be wasted searching in the wrong direction. However, the heuristic function of the A* algorithm fixed the problem since it prioritizes the nodes going in the right direction. Moreover, another consequence of its unfocused search is that the optimal path is not always found by Dijkstra's algorithm when weights or distances between nodes are the same.

Therefore, it is clear that the A* algorithm is usually faster than Dijkstra's. The Figure 2.1 shows the comparison between both algorithms in terms of the search focus.

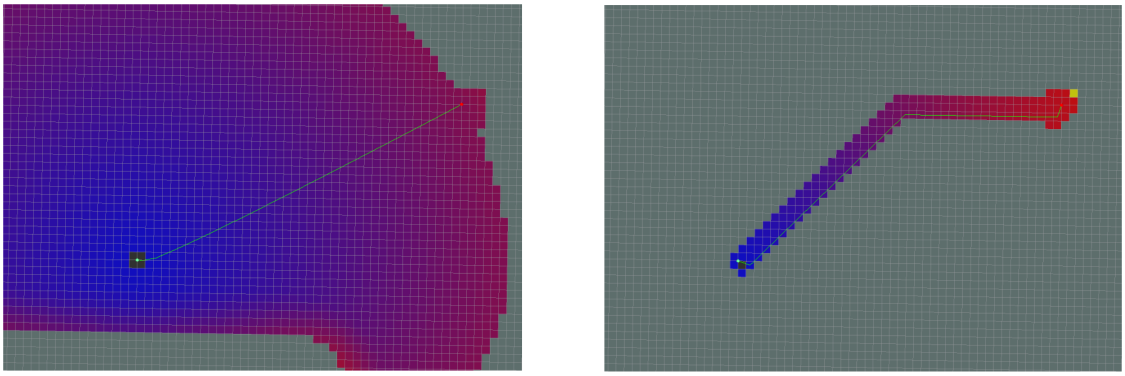


Figure 2.1: Comparison between Dijkstra's (left image) and A* algorithms (on the right) [36].

In relation to the A* algorithm, a possible drawback can be the heuristic algorithm since it depends on its accuracy to compute the function $h(n)$.

Finally, a limitation can be encountered in both algorithms complexity-wise. For instance, in large and high-dimensional maps, they are computationally intensive, despite A* being a big improvement over Dijkstra. Though, there are numerous alternatives of, mainly the A* algorithm adaptations, regarding global planners, that present some advantages over the original ones.

2.1.1.2 ROS Planners

NavFn

The NavFn [37] global planner is based on the NF1 navigation approach [38] and provides a fast interpolated navigation function to create plans. Assuming that the robot has a circular shape, the inputs for computing the minimum cost plan are the costmap, starting point, and end point. Nonetheless, it only uses the graph-based search Dijkstra algorithm, the developers expect to add the A* algorithm in the future.

The disadvantage of NavFn is that the obstacle circumvention does not take into account the robot's motion characteristics and footprint, only the shortest straight path, meaning collisions can still occur [39].

Global_Planner

The global_planner [36] is the replacement of NavFn Global Planner as it implements a more flexible solution of the interpolated function used to create plans. This planner supports both Dijkstra and A* algorithms and has a slightly different approach to calculating the potential fields compared to the previously presented planner. It also includes an option to reproduce the behavior of the old NavFn global planner and an orientation filter as a post-processing step to decide which orientation the robot has along the path.

2.1.2 Local Planners

According to the global plan, local planners assure the robot follows the trajectory generated, dynamically avoiding previously unknown obstacles present in the environment, i.e., not represented in the initial costmap. Local planners, as global planners, use waypoints to create collision-free paths. Considering the information (waypoints) from the global planner together with sensors' data, a limited local path is generated at a specific rate, using only portions of the map surrounding the robot. Thus, local planners try to update the map cyclically, creating a collision-free path as close as possible to the global trajectory considering known and unknown obstacles.

Some of the most used local planners are the Dynamic Window Approach (DWA) and Time Elastic Band (TEB), which are presented in the next sections. They are not exclusively from ROS, since there are many implementations outside of the framework as path planning solutions for other applications besides robotics.

2.1.2.1 ROS Planners

Dynamic Window Approach (DWA)

The Dynamic Window Approach planner algorithm [40] relies on the concept of a rolling window in the surroundings of the robot calculated according to the kinematics model and the current speed of the robot. The trajectories are generated for each set of speeds within the window range. Then the optimal speed is obtained by estimating the trajectories based on a certain evaluation function which generally considers three factors: speed, path angle, and distance from the obstacle (obstacle clearance).

The main advantage of this planner algorithm is that numerous paths are calculated (predicted), that is, rolled out, and then the optimal one is chosen based on some important factors or constraints related to the robot, its state, and the environment. This behavior makes the planner suitable for different types of locomotion, such as differential and omnidirectional [41], and environments both static and dynamic. However, is

not suitable for car-like robots since it does not calculate solutions considering motion reversals. Moreover, another disadvantage of this planner is the high computational demand due to its predictive behavior. The Figure 2.2 shows the DWA planner's predictive behavior.

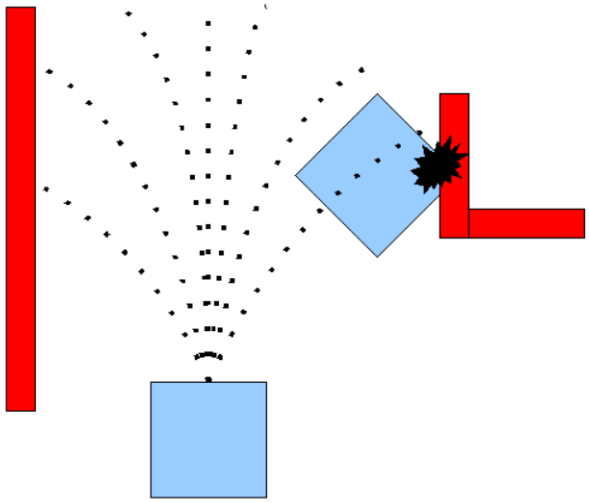


Figure 2.2: Dynamic Window Approach planner's behavior [42].

Elastic Band (EB)

The Elastic Band (EB) Planner is based on the Elastic Band approach [43]. As the name suggests, the concept relies upon the deformation of the path generated as an elastic band. There are two forces applied to the path: an internal contraction force which simulates the tension in a stretched elastic band, and an external repulsive force which confers a repulsive behavior from obstacles. Thus, these two forces in equilibrium mimic the nature of an elastic band.

This type of behavior gives the planner the ability to handle dynamic situations with moving obstacles since the two forces deform the path in real-time, always reaching new equilibrium positions as the robot perceives new unknown obstacles.

Furthermore, the standard Elastic Band approach resides on the concepts of the path's deformation generated by the global planner, avoiding obstacles in the way and minimizing its length. However, this approach is not considering the mobile base motion constraints, leading to possibly not perfectly suitable paths for the type of robot being used, being one of the major limitations of the EB planner.

Time Elastic Band (TEB)

Time Elastic Band (TEB) Local Planner is based on the same approach as the last planner.

The TEB approach improves the prior by taking into account the constraints of the robot being commanded for motion planning. Contrary to Elastic Band's planner, it considers temporal aspects of motions for a more realistic approach to path planning. It also contemplates kinematic constraints, such as linear and angular velocity, acceleration, and, as the EB planner, the trajectory generated derives from the deformation of the global path. As the main objective, TEB wants to reach a goal in minimal time, following a collision-free path in order to adhere to the kinematic and dynamic constraints of the robot. The TEB approach also maintains and optimizes a set of candidate trajectories in parallel. This way, it can change between the candidate trajectories, choosing the best current globally optimal trajectory.

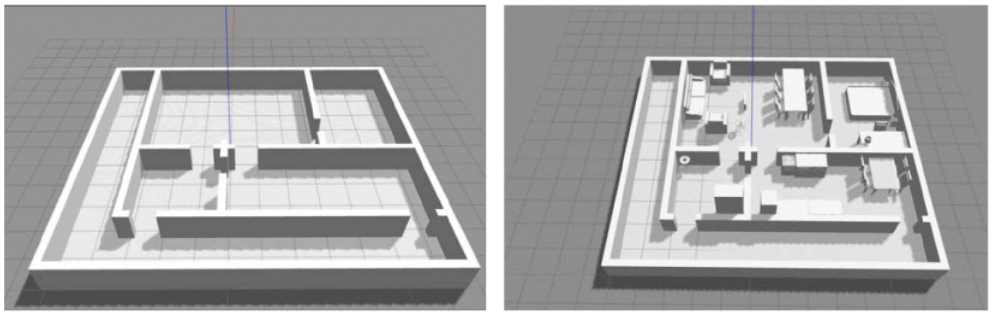
One of the major disadvantages of TEB is that, the repeated calculation of a new path at every cycle causes a big computational overhead, and, as a result, it needs a powerful computer in order to meet such computational demand.

Discussion

Pimentel *et al.* have conducted two studies [44] [45], in mobile robot navigation using different ROS planners. The first [44], evaluated the navigation performance in a static environment with and without objects, and several tests were performed with different types of sensors and combinations: only the front laser scanner, both front and back laser scanners and also, and the front and back laser scanners combined with a 3D camera. It is clear that the performance of the planners was consistent and dependent on the type of environment and not the number of sensors. That is because the results' consistency depends on the representation of the environment, despite the number of sensors. However, sensor redundancy is a strategy to mitigate the noisy data produced by some, not affecting the planners' performance.

Thus, the tests' results, with different sensors' combinations, were similar in each scenario (Figure 2.3). EB planner was the best in scenario one, and TEB was the best in scenario two. The DWA planner was the worst in both scenarios. The study also concluded that EB Planner was the best for social navigation as its performance is the most balanced in both scenarios and independent of the sensors used.

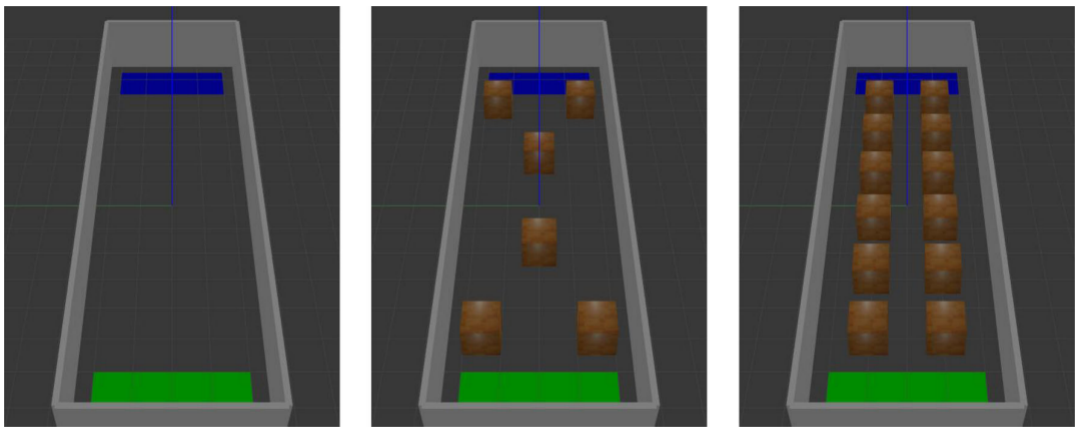
In the second study [45], the tests targeted both global and local planners and were divided into six stages with multiple configurations and scenarios, but only four of them are important for this discussion. The *stage one* was characterized by a simple configuration, and the scenarios used had only easy-to-perceive obstacles or no obstacles at all. The main focus of the *second stage* was to evaluate the performance of global planners. The tests were executed in the same scenarios as the previous stage, however, the localization



(a) Environment 1 (b) Environment 2

Figure 2.3: Scenarios used for EB and TEB planners' comparison [44]. (a) Environment without obstacles (b) Environment with static objects.

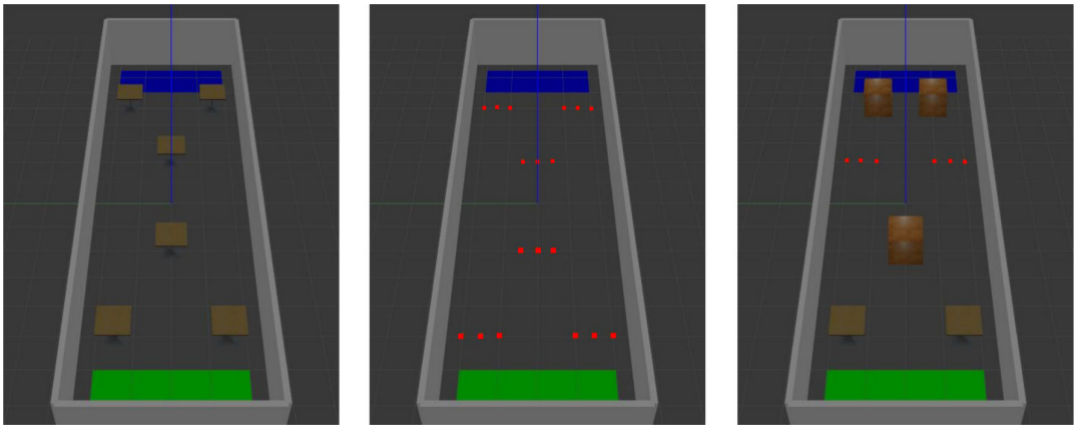
source was changed (to the localization given by the simulator, in contrast to the previously used Adaptive Monte Carlo Localization (AMCL)), as well as the global planners (to Navfn and Global Planner). In relation to the *stage three*, the Navfn was chosen as the global planner due to the best results and different local planners were tested (*Trajectory Planner*, *DWA*, *EB* and *TEB*) while the test scenarios stayed the same. In the *fourth stage*, more difficult obstacles (hard to see by laser scanners) were introduced into the environment. TEB was selected as the local planner, and different sensors' observation sources were tested. The test scenarios and corresponding stages are depicted in Figures 2.4 and 2.5.



(a) Simple (b) Boxes (c) Narrow Corridors

Figure 2.4: The simulated environments used in stages 1, 2, and 3. The green area represents the start region, and the blue area represents the goal region [45].

In the simple configuration (stage one), the results showed that the best performer overall was the EB planner. In the fourth stage, where the environment was dynamic with people walking and human interaction, TEB was the winner, but EB's performance



(a) Tables (b) Boxes Small (c) Mix 1

Figure 2.5: The simulated environments used in stage 4. The green area represents the start region, and the blue area represents the goal region.

was close to TEB's. Regarding DWA planner, it was worse than the others in all stages. Although it is very popular among the robotics community, nowadays, there are better solutions to address new navigation problems, such as elastic band theory-based planners[46] [47].

From both studies, evidence shows that EB planner was the most balanced planner considering both static and dynamic scenarios and is less computationally demanding than TEB planner. Nonetheless, TEB local planner was the most adequate and natural for dynamic environments. Finally, the second study also suggests that it has higher precision in narrow environments.

Regarding highly dynamic environments, unpredictable and unexpected situations have a high chance of happening. Therefore, robots have to be provided with recovery mechanisms that help them to make decisions when encountering new situations and to know how to behave. The next section describes the inner mechanisms of such behaviors and how they function.

2.2 Behavior Trees

Behavior Trees (BTs) was first introduced by the necessity of creating Non-Player Characters (NPCs) in the gaming industry, who can switch between different tasks depending on the action of the main character. They appear in popular gaming engines like Pygame and Unreal Engine. Before them, Finite State Machines (FSM) was the first widely used solution for decision-making processes. They are based on *one-way control transfer*. That is, the base code behind every scheme is executed sequentially with no return [10]. This behavior raises a limitation: a tradeoff between modularity and reactivity.

In reactive systems, multiple scenarios must be considered, which, in terms of the implementation itself, means many transitions between states. Therefore, removing a component of the system creates the necessity of revising and refactoring it. Consequently, FSM does not provide much modularity to build complex systems and be easily reused and adapted. The following Figure 2.6 shows an example of a FSM.

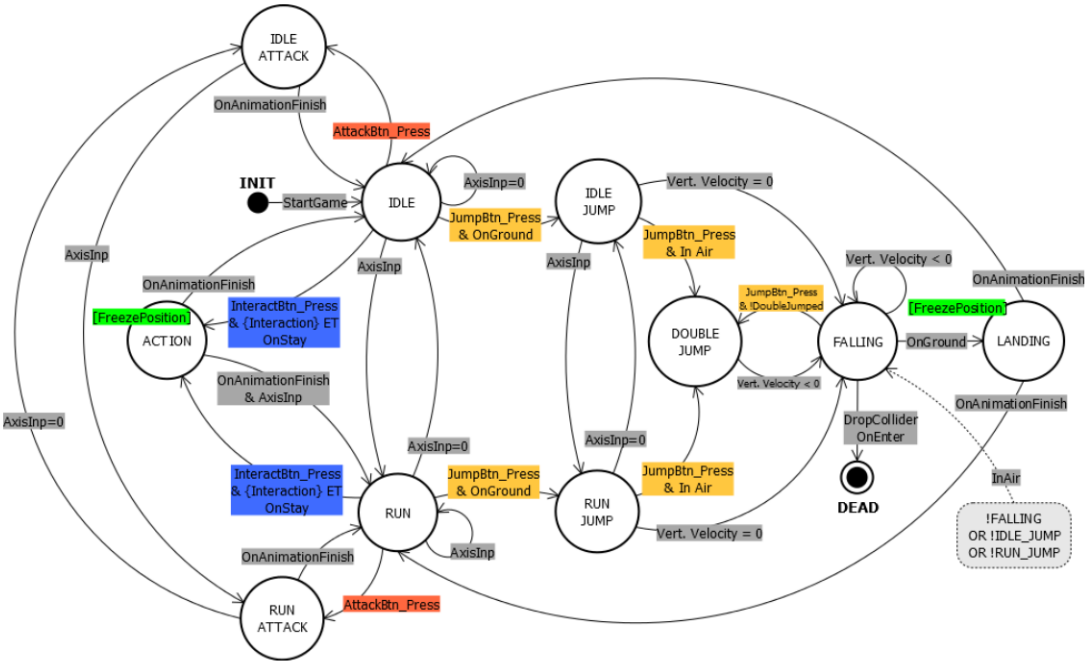


Figure 2.6: Example of a FSM. States are characterized by circles. The text near arrows symbolizes actions and conditions [48].

Before introducing the advantages of BTs, is important to clarify the concept. A BT is a directly rooted tree which has four control flow nodes - fallback, sequence, parallel, and decorator - and two execution nodes - action and condition. The execution of the nodes in the tree is controlled by a signal called *tick*[1] that is propagated throughout the tree. The root initiates the cycle from left to right, ticking its children, and they have three possible answers to return to its parent node. They can return *running* meaning the execution of the node is still in progress, *success* in case they successfully achieved their goal or *failure* in case of no success. The following subsections explain the different types of nodes.

2.2.1 Control Flow Nodes

Fallbacks

Fallbacks, also called Selector nodes, represent the versatility BTs can have. The fallback node provides different ways to achieve the same goal. Each fallback item is

[1] *Ticks* are generated signals with a given frequency to ensure synchronization of the system's nodes

executed only upon the *failure* of the previous one, meaning that if the first child returns success, the node returns success as well. Thus, the next children are not ticked because the previous one had already returned *success*. However, if a child is not successful, the next ones are executed, from left to right, until one returns *success*. This type of behavior opens a range of possibilities for applications and increases the flexibility of the system. The fallback node is represented graphically by a question mark (?). The Algorithm 3 clarifies how the fallback node works.

Algorithm 3: Pseudocode of a Fallback node with N Children

```
1:  Function Tick() :
2:  for i ← 1 to N do
3:      ChildStatus ← child(i).Tick()
4:      if ChildStatus == Running then
5:          return Running
6:      else if ChildStatus == Success then
7:          return Success
8:      else
9:          return Failure
10:     end if
11: end for
```

Sequences

Sequences are a set of dependent actions that need to be performed in order. Therefore, the *success* or *failure* of one child, directly influence the entire sequence. In contrast to the fallback node, the *success* of one child is required in order to move to the next one. If one child returns *failure*, the sequence fails. On the other hand, if all the children are successful, the node returns *success*. The sequence node is usually represented by an arrow (->). The Algorithm 4 shows how the sequence node works.

Algorithm 4: Pseudocode of a Sequence node with N Children

```
1:  Function Tick() :
2:  for i ← 1 to N do
3:      ChildStatus ← child(i).Tick()
4:      if ChildStatus == Running then
5:          return Running
6:      else if ChildStatus == Failure then
7:          return Failure
8:      else
9:          return Success
10:     end if
11: end for
```

Parallel

Parallel nodes (Algorithm 5) obey to a rule of execution. Considering N the total number of children and that all children are ticked, a parallel node returns success if M children return success, failure if $N - M + 1$ returns failure or running otherwise. Parallel nodes are mostly used in probabilistic behaviors, and they are represented by two arrows.

Algorithm 5: Pseudocode of a Parallel node with N Children

```
1:  Function Tick() :
2:  for i ← 1 to N do
3:      ChildStatus[i] ← child(i).Tick()
4:      if ∑_{i:ChildStatus[i]==Success} = M then
5:          return Success
6:      else if ∑_{i:ChildStatus[i]==Failure} > N − M then
7:          return Failure
8:      else
9:          return Running
10:     end if
11: end for
```

Decorator

Decorator nodes are control flow nodes with only one child, and they change the return value of its child or *Ticks*[1] it, according to a defined rule. Some examples of decorator nodes are the *inverter* which does as it's called, it inverts the return value of its child, the *max-N-tries* which only lets its child fail N times before return *Failure* and the *Timeline* which *Ticks* its child nodes for a limited amount of time.

2.2.2 Execution Nodes

Action

An action node is set to perform some operation when it receives *Ticks*[1]. The node returns *Success* if the operations are completed and *Failure* if it fails to complete them. Otherwise, it returns *Running*. During the *Running* phase, if the node stops receiving *Ticks*, it aborts its execution. of an action node. An action node is represented graphically by a rectangle, and the Algorithm 6 shows an example of it.

Condition

When a condition node receives a *Tick*, it checks whether the condition is satisfied and returns *Success* or *Failure* accordingly. Algorithm 7 exemplifies the condition node, and it is represented graphically by a circle.

Finally, the following figure (Figure 2.7) summarizes all the possible standard nodes used to create BTs.

Algorithm 6: Pseudocode of an Action Node

```
1: Function Tick() :
2: ExecuteAction()
3: if action – succeeded then
4:     return Success
5: else if action – failed then
6:     return Failure
7: else
8:     return Running
9: end if
```

Algorithm 7: Pseudocode of a Condition Node

```
1: Function Tick() :
2: if condition – true then
3:     return Success
4: else
5:     return Failure
6: end if
```

Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	◊	Custom	Custom	Custom

Figure 2.7: Type of BT nodes [10].

2.2.3 Discussion

In the beginning, BTs brought a new and easier way to build robust and complex systems which can adapt and behave in different circumstances, that is, being capable of making decisions, easily reusable, and modular to address numerous applications. Therefore, in the last decade, the robotics community showed interest in the benefits of BTs to develop more robust, complex, and modular robotics and autonomous systems[49].

Contrary to FSM, BTs are based on *two-way control transfers* which is the way that most modern programming languages work: using function calls. Functions are called by the main code for the execution of specific tasks and return to it to continue the code's execution. BT nodes have the same behavior where higher-level nodes can represent a global task (main code) which is divided into smaller ones (functions) that directly influence the success of the global task. Consequently, the main important advantages of these types of structures are [49]:

1. **Task Hierarchies**. The idea of task hierarchies is common when it comes to modular systems. For example, different agents in multi-agent systems are managed by a coordinator which defines a higher hierarchical level than the agents[50]. In case of BTs, there are tasks composed of subtasks which, consequently, can have their own subtasks as well. For instance, the Figure 2.8 represents an example of a task using a BT - a player decision-making process (represented by the first fallback node) - with subtasks - its behavior in different scenarios (another fallback node and sequences). First, if any opponent player is visible, he just wanders around. However, if there's an opponent in sight, he goes to battle mode, which is divided into three actions: firing a gun, swinging a sword, or taunting the opponent player. Moreover, firing a gun implies that more than three sub-actions are taken.

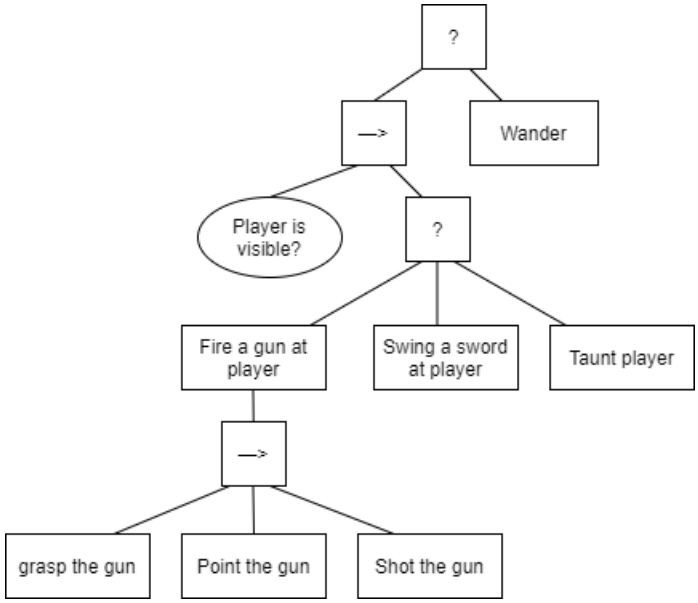


Figure 2.8: Example of a BT of a game (adapted from [49]).

2. **Reactivity** implies that the system is capable of interrupting less important tasks in order to execute the more important ones. Prioritization is one of the key components of a robust and autonomous system. For example, a robot system has to prioritize the amount of battery left more than almost any other task. Hence, in case of a low battery, the robot should interrupt the current task and start its routine to search and navigate to a charging station.

3. **Modularity** is the biggest advantage of BTs over FSM. It is achieved by having the same interface for every node. Each node returns *success*, *failure* or *running*, which is enough for higher tasks to decide where the system evolution's direction. For instance, the fully modular ROS2 Navigation Stack [51] (Figure 2.9) is built by BTs. The recovery, controller, and planner server are managed by a higher entity - the BT Navigation Server. These completely modifiable servers show clearly that BTs can

make can give a system the ability to be modular and reconfigurable. Thus, regarding ROS2 Navigation Stack, each developer can use their own recovery, controller, or planner server as they can be fully replaceable.

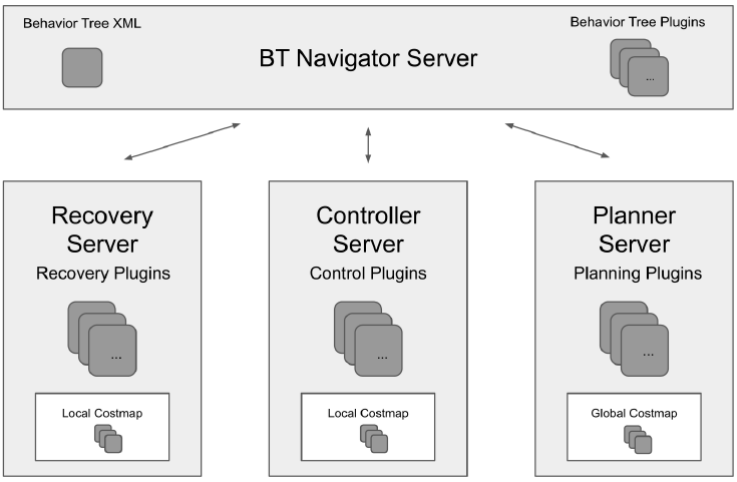


Figure 2.9: ROS2 navigation stack scheme[51].

However, there are disadvantages to be highlighted when it comes to the complexity and performance of BTs.

Behavior Tree based applications can be complex to implement. As the synchronization of the system is maintained by tick's[1] generation, it has to happen in parallel with the actions being executed, therefore implementing BTs with single-threaded programming can be a challenge[10] as some tasks can be performed asynchronously.

Moreover, the creation of different alternatives that BTs provide comes with a performance cost, especially in closed-loop task execution[10]. In smaller systems, the cost can even surpass the advantages. Consequently, they become too slow or unfeasible for their purpose and don´t have any advantage compared to other more simple architectures. For example, when robots operate in very structured, simple, and easy-to-predict environments.

Finally, BTs software tools are in the early stages of development compared to others, for instance, FSMs. However, despite the aforementioned drawbacks, the robotics community has been developing more tools in recent years which take advantage of BTs. *Related work* about BT applications in robotics revealed a mid-stage development. Impactful projects were developed or under development and reveal that they can be versatile and easily adapted across different applications. Their modularity, flexibility, and ability to create complex reactive systems and decision-making processes have made the adoption of these structures beneficial to the progress of robotics research and the creation of new, improved, and efficient solutions. Among them are *manipulators* [52], [53], *aerial robots* [54], [55] and *wheeled robots* [56], [51]. The majority of BTs studies on ground

robots show that they are useful for the robot's control system, helping decision making, obstacle avoidance, and multi-task switching in industrial environments and search and rescue scenarios.

2.2.4 Behavior Trees Open-Source Libraries

In this section, the most used BT Open-Source Libraries will be analyzed and compared with respect to important characteristics. Thus, regarding the development of behaviors using BTs, the important characteristics considered are:

- The language;
- The compatibility with ROS;
- The existence of proper documentation;
- The existence of a GUI to visualize the developed BTs;
- Code maintenance and development.

The Table 2.1 summarizes the BT libraries and the chosen characteristics.

Table 2.1: BT Open-Source Libraries identified (adapted from [57]).

Name	Language	ROS	Doc.	GUI	Last Commit
BehaviorTree.CPP	C++	yes	[58]	yes	02-02-2022
py_trees	Python	no	[59]	no	01-12-2021
py_trees_ros	Python	yes	[60]	yes	10-05-2021
CoSTAR	C++	yes	[61]	yes	10-09-2018
UE4 Behavior Tree	UnrealScript	no	[62]	yes	N/A

From the six chosen BT libraries for the comparison, there is a distinction to be made: five of them were developed for the purpose of robotics (highlighted in bold) and one for games. Therefore, the analysis will not cover *Unreal Engine 4 (UE4)* BT library since the development is focused on modeling intelligent behaviors for characters in games. Hence, the criteria to classify a well-developed and structured BT library for games cannot be the same since these two fields prioritize different characteristics. For instance, event-driven programming is the major concern in the gaming industry, whereas robotics is time-triggered control [57]. Although, it is important to mention that *UE4* is one of the famous engines for game development, well documented and actively developed.

BehaviorTree.CPP

BehaviorTree.CPP is one of the most used BTs libraries to develop applications using BTs. Although the main use case is robotics, it can be used to build AI for games [58]. The library is written in C++ 14 and trees can be written in a *scripting language (based on*

Extensible Markup Language (XML) and loaded at run-time. Nodes can also be converted into plugins and loaded at run-time as well. This is an important feature to address as the performance of ROS and its structure benefit from code that can be loaded as plugins. Therefore, BehaviorTree.CPP is fully compatible and easily integrated with ROS.

Using the date that the libraries were checked (04-02-2022) as a reference, BehaviorTree.CPP was the most up-to-date (02-02-2022) with 847 commits and eighty contributors, signaling that is being actively developed to date. It is also very well documented [58] with a considerable number of tutorials and detailed information. Finally, this library has a GUI interface called *Groot*, which makes easy the debug, monitoring, and visualization processes. However, according to the authors, Groot is not actively supported by them.

Py_Trees and Py_Trees_Ros

Py_Trees is the main BT library used by the Python community [57]. However, since it was not developed to work with ROS directly, but in robotics in general, it was added Py_Trees_Ros, because it is an extension of the former which works with ROS. Regarding composite nodes, it includes sequence, Selectors, and Parallel nodes. To date, the fallback node is not implemented.

Py_Trees and Py_Trees_Ros libraries, are actively supported as their last commit date was 01-12-2021 and 10-05-2021, respectively. Both have more than 1000 commits to date and between ten to twenty contributors. Although the number of contributors is slightly less than the previous library, this library is considered actively supported and in development. They are also well documented as the available resources and tutorials [59], [60] are many. According to the documentation, Py_Trees does not support a GUI, however, Py_Trees_Ros does.

CoSTAR

As stated in [61], *CoSTAR is an end-user interface for authoring robot task plans*. CoSTAR aims to facilitate users in programming robot tasks for different use cases. It integrates perception and planning in a BT base user interface.

It is a library implemented in C++, and it works with ROS. However, according to the documentation, it only supports ROS Kinetic and Indigo versions. Regarding nodes, Sequences, Fallbacks, and Decorator nodes are implemented, however, with a different categorization such as Service, Variable, Logic, Action, Query, and Condition Nodes [53]. This library has a beautiful BT based GUI with several boxes depending on the type of node and action pretended. The documentation is not as abundant as BehaviorTree.CPP and Py_Trees, but are carefully made. However, the last commit date (10-09-2018) leaves the perception that the platform is not being actively developed, despite having about 1900 commits and ten contributors.

2.2.5 Behavior Trees Libraries Discussion

In this section, the aforementioned will be compared, and the advantages and disadvantages of each one will be outlined.

Studies show [57] that in Py_Trees, decorators were rarely used in robotics projects, constituting only 6% of the composite nodes (Decorators, Parallel, Fallback, and Sequence). That is because in Python code is much easier to apply transforming operations than using the BT abstract syntax. However, the same thing cannot be said about BehaviorTree.CPP library as decorators are used almost three times more (19% of the composite nodes). The benefit of having a GUI is clear in this case, as decorators can be visualized, monitored, and controlled. In Py_Trees, the nonexistence of this type of interface makes it harder to use certain types of nodes since debugging is difficult and tedious in case of some errors occur.

Moreover, the presence of a dedicated XML file to program the BTs, in the BT library BehaviorTree.CPP comes with a structural advantage: the rest of the system can read and write from the backboard when BTs can be fully written in another independent file. This separation allows BTs models to be processed in other systems. Therefore, BehaviorTree.CPP developers thought about one of the keys of BTs, their modularity.

Nevertheless, in the study conducted by Ghzouli *et al.* [57], 30% of the projects analyzed belonged to BehaviorTree.CPP, whereas 70% was from Py_Trees_Ros. The suspicion is that the reuse by reference through file inclusion is harder because it has to be made at the source code level, and the BTs are declared in a separate XML file rather than in the latter, as BT models are intertwined with the Python code.

Finally, compared to BehaviorTree.CPP and Py_Tree_Ros, CoSTAR's Graphical User Interface is the best, by its structure, functionality and ease of use by users. However, CoSTAR seems like a all-in-one integration tool. It has perception, and path planning were not needed in a BT tool as ROS has those packages integrated. Moreover, the fact that only old versions of ROS are supported and the source code is not actively maintained closes the doors for more improvement and for attracting more people into the project.

3

Supporting Concepts

In this chapter, supporting concepts to understand the work developed are outlined. That is, the Robot Operating System (ROS). In Section 3.1, the architecture of ROS is explained so the reader can understand its inner mechanisms and communications with some examples supporting the presentation of such concepts.

3.1 ROS Concepts

Robot Operating System (ROS) is an SDK (software development kit) that provides the building blocks needed for developers to build robot applications [63] which have been widely used, over the years, in numerous applications such as research projects [64], [65] and real-world implementations [66], [67], [68]. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and numerous other tools [69] to help developers to achieve their goals faster.

ROS architecture (Figure 3.1) is based on the Publish-Subscribe paradigm where processes, called *nodes*, communicate with each other through *topics* and *services*. Moreover, as ROS is not a real operating system[70] [63], despite the name, and it runs on Linux (Ubuntu). ROS was created to build distributed systems in which each block in the system doesn't depend on each other, however, this goal was only achieved with the second version of ROS, ROS2 [71]. In the first version of ROS, the small programs (nodes) that are saved inside packages (building blocks) run in a distributed way, but they all depend on *roscore*, which turns the solution fairly centralized.

3.1.1 ROS Core Elements

The core of the ROS involves two parts: the *ros master* and the *parameter server*. *Ros master* node is basically a database of all the topics, nodes, and services available on the system, including their names, message types, and configuration files, as well as their locations (packages). The *parameter server* works as a global list of settings for the nodes. It carries all the values of the variables used by the nodes in the network. As they are public, all the different nodes have access to the same list of settings.

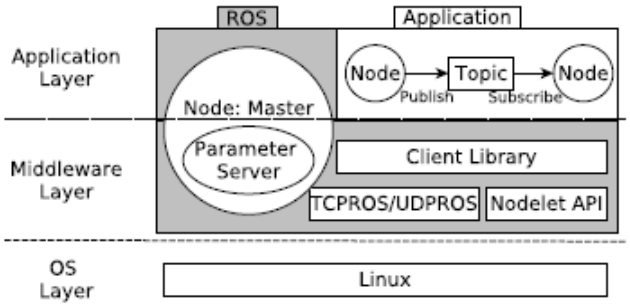


Figure 3.1: ROS architecture[72].

3.1.2 ROS Nodes

ROS nodes are small programs written to execute one or more tasks. They are intended to interact with other nodes, provide and gather the necessary information from the network - specifically from topics - to reach their goal. Nodes are the kernel of the robotic system. For instance, one node is responsible for analyzing the laser scanner's data from the robot's sensors in order to avoid obstacles, another node controls the robot wheels, and another performs the localization of the robot in the environment. They communicate with each other using *topics*, *services* and *actions*.

ROS nodes are created inside a special folder that is called package. Each package can have multiple nodes communicating with each other and other nodes outside the package. Moreover, each version of ROS has its standard packages, which are used by all robotics developers. For example, the package *move_base* implements an action that, given a goal in the world, will attempt to reach it with a mobile base [73]. Basically, this package provides the implementation of the architecture for any robot to move from one point to another in space. Moreover, inside the move_base, there are other packages that perform specific tasks. Each is a building block of one architecture, such as the global and the local planners, implementation of costmaps and localization, and recovery behaviors.

Therefore, it is paramount that ROS developers follow the default rules and best practices so that each package can be easily updated, adapted, or changed in order to meet the needs of different systems.

3.1.3 ROS Communication Tools

There are three ways of communication in ROS: via *topics*, *services* and *actions*, using standard *messages*.

- **Topics** are the main form of interaction in ROS. Topics are endpoints for data streams to be exchanged by nodes, actions, and services. The communication process starts when an entity wants to publish or gather information on a topic. The entity communicates to the master the exact port it deserves to open. Then a response

is received from the master server with a list of the current open ports. Thus, the requests for opening ports and closing ports of communication in the systems are handled by the master server. Once information is published in a port (topic), other nodes that need it to listen to that port to gather the right information to perform their tasks. Nodes can create entities called *publishers* and *subscribers* according to their intent, for instance, the direction of the data. If they want to gather information, they create a subscriber to listen to a topic, whereas the intent is to provide information to the network, they create a publisher. Each publisher and subscriber have to mention the name of the topic they want to subscribe to or publish. Finally, ROS nodes can have multiple publishers and subscribers according to their complexity. The process of communication via topics is illustrated in Figure 3.2. In the example, the node called */amr/localization/map_server* publishes information in the topic called */amr/localization/map*. Then the the node */amr/localization/lama_loc* subscribes the topic */amr/localization/map* which means it is receiving messages from the */amr/localization/map_server* node.

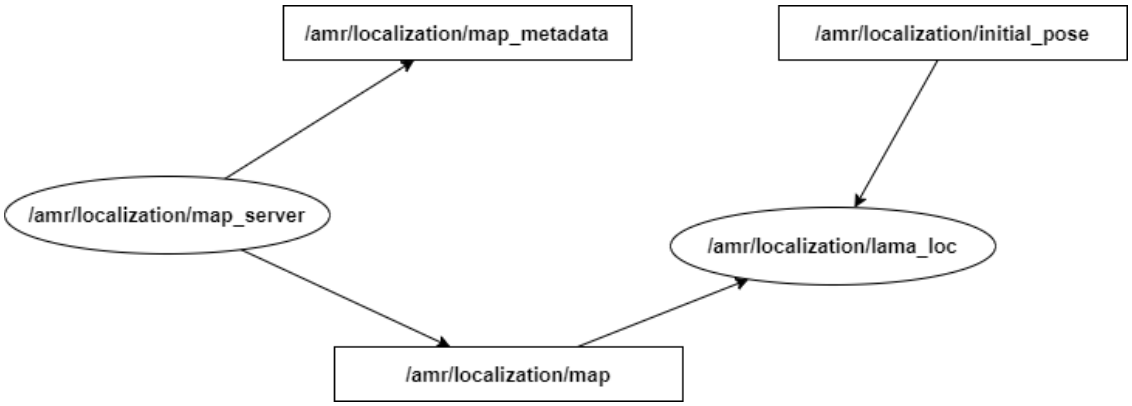


Figure 3.2: Communication example between ROS nodes. Nodes are represented by circles, whereas topics are represented by rectangles.

Security-wise, ROS does not provide any restriction about the integrity of the data being published, therefore, there is no access control regarding topics.

- **Messages**. Information in ROS is structured in different message's types. ROS messages are a standard way of organizing information, so all the systems follow the same rules. Thus, each topic, service, or action has a message type associated with it, so rules for communication have to be respected. For example, node A (publisher) sends data to a topic of the type sensor_msgs/Laserscan. The only way node B (subscriber) can listen to the information published by node A is by accepting to receive that message's type. Therefore, the communication of each set of publishers/subscribers is only possible if they all use the same message type. In Figure 3.3 is illustrated the aforementioned example and Listing 3.1 shows the message *geometry_msgs/Pose* type which includes in its structure the *geometry_msgs/Point*

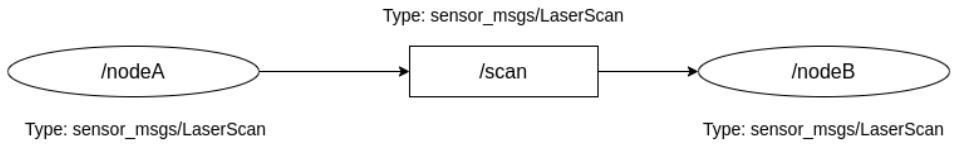


Figure 3.3: ROS nodes communicating via the topic scan.

type, describing a position in space and the *geometry_msgs/Quaternion* type, describing an orientation. Moreover, the types geometry_msgs/Point and geometry_msgs/Quaternion are described by three and four numbers with the float type. Finally, having standard message types increases modularity and development efficiency. There are numerous ROS standard types of messages, although developers are free to create their own types if it suits their applications.

Listing 3.1: Structure of ROS Geometry Messages Pose type.

```
#geometry_msgs/Pose

geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

- **Services** provide an easy way of establishing synchronous client/server communications. The client sends requests to the server, and the server responds by giving feedback to the client. They are usually used for simple tasks that don't require acknowledgment of completion, such as enable/disable, set/unset or request for data. For example, in a robotic system, it is required to enable/disable some sensors to perform some task or are causing a decrease in performance (laser scanners or RGB cameras), ask for information about the robot's location at a given time or trigger an action (start charging after docking). Although multiple service clients can use the same service, only one service server can exist for a given service. In Figure 3.4 and Listing 3.2 are shown the client/server communication data flow and the structure of a ROS service's declaration, respectively. A ROS service declaration have two two fields: the request and response variable(s) name(s) and type(s), separated by three dashes. In Listing 3.2, the client has to send an integer in the request message, and the response message received has two variables: one boolean, which is meant to give the success state, and a string with a message. Nonetheless, as ROS messages, new ROS services can be created by the developers to meet the needs of their applications.

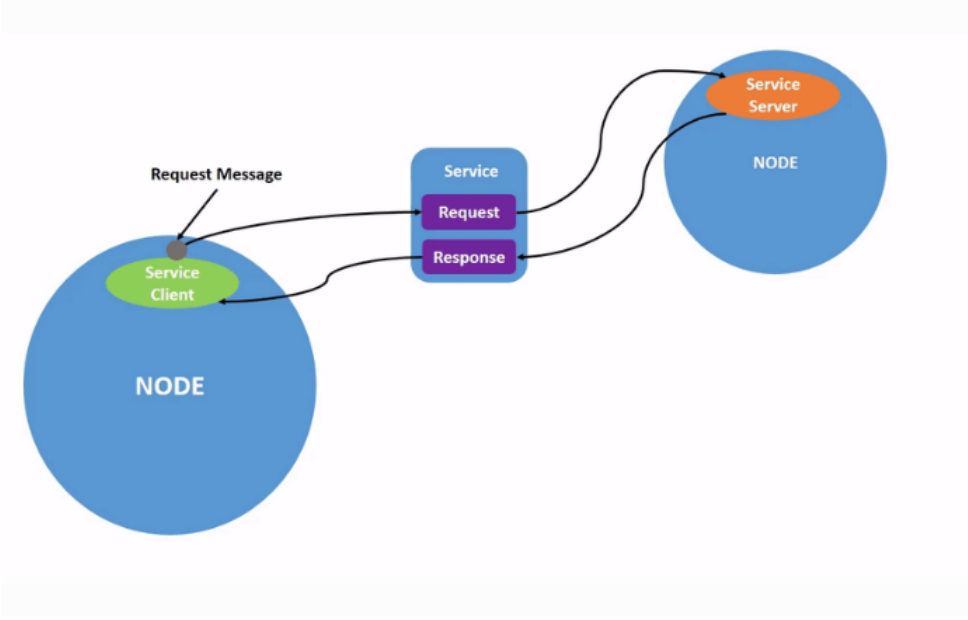


Figure 3.4: ROS Service communication structure [74].

Listing 3.2: ROS Service declaration structure.

```
int8 value # node id
---
bool success # returns success or failure
string message # in case of failure returns the error message
```

- **Actions**, similarly to services, are based on a client/server communication approach, however asynchronous. They are usually implemented to execute more complex tasks with a longer duration. Consequently, they are asynchronous since blocking the system during a long period of time can have catastrophic consequences in some applications. Hence, the client can access the active state at any time, monitor the process, and, in case of need, cancel or halt it. For instance, a client makes a request for the robot to go to a location in space, but suddenly a motor stops working. In that case, the action has to be canceled, and the robot starts the emergency procedure. Similarly to ROS services, ROS actions have a request message and a response message. However, as they execute processes with a longer duration, they provide feedback during the execution. Thus, ROS actions include two ROS services, the goal service, the response service, and a feedback topic. Moreover, depending on the task of each entity and the name of the action, there is a type of message associated with it. For instance, in ROS navigation stack, there is an action called *MoveBaseAction*. Thus, the type of messages associated with the goal service, response service and feedback topic are *MoveBaseActionGoal*, *MoveBaseActionResponse*, and *MoveBaseActionFeedback*.

In Listing 3.3 and Figure 3.5 are shown the communication data flow and a declaration structure of a ROS action, respectively.

Listing 3.3: ROS Action declaration structure.

```
#goal definition
geometry_msgs/PoseStamped goal
---
#result definition
bool result
---
#feedback definition
float64 current_velocity
```

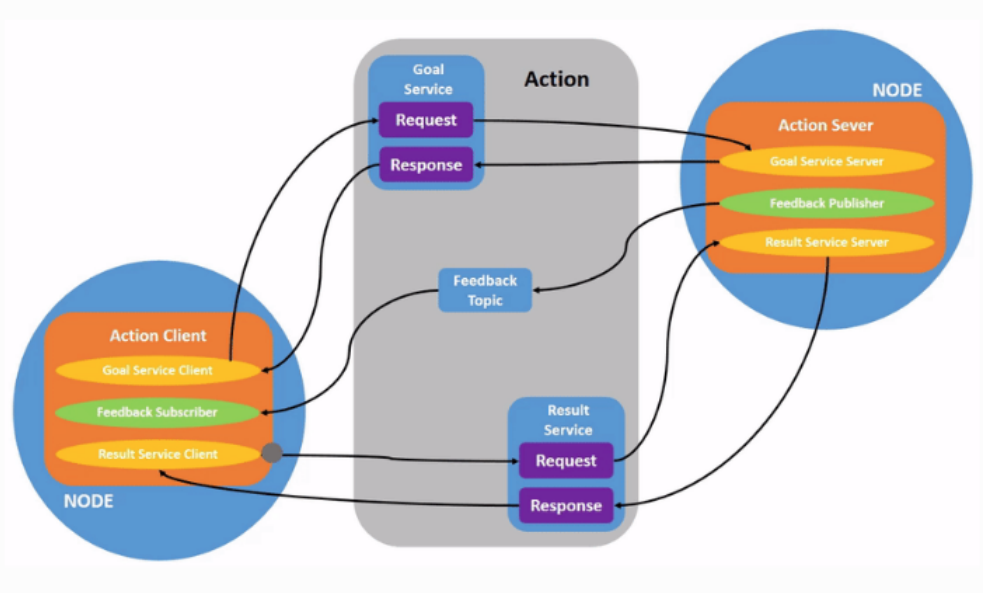


Figure 3.5: ROS Action communication structure [75].

4

Architecture and Implementation

In this chapter, the implemented architecture is explained. After a general overview, each important block of the architecture is described in detail, their dataflows, and each main algorithm. Section 4.1 presents the overview of the architecture, whereas the Section 4.2 details the role of each layer and shows the dataflows and algorithms in them. Finally, the Section 4.3 presents the created robotic behaviors.

4.1 Architecture Overview

The implemented architecture follows similar principles as the research work led by *colledanchise et al.* [76] and which presents a system divided by three separated layers organized by order of abstraction: the *Task Layer*, the *Skill Layer* and the *Service Layer*. In the architecture herein proposed, the name and the implementation changes from *Service Layer* to *Interface Layer* and practical differences will be detailed further ahead in the next sections.

The first layer possesses the most conceptual abstraction, whereas the last holds the least. Conceptual abstraction means broadness of control within the system. Particularly, the Task Layer knows what the system will do but doesn't know how. It just sends to the layers above an order (task) without knowing how it will be done. Progressively, the task will be broken down into small processes as the level of abstraction reduces. The Figure 4.1, demonstrates the communication between layers. The Task Layer sequentially sends an *Execute Skill* orders to the Skill Layer (middle-level layer) with the purpose of executing the behavior.

The Skill Layer is where the skills are declared and executed. A behavior is a set of skills organized together to produce an outcome. Thus, it is in the Skill Layer where the basic capabilities of the robot can be identified, for instance, grasping an object or moving to a position. Consequently, skills are broken down into small requests to the Interface Layer, which are necessary to complete them successfully. The Figure 4.1 shows that service and action requests are performed between the Skill Layer and the Interface Layer.

Finally, the Interface Layer has the necessary interfaces to communicate with the outside entity, which can communicate directly with the robot's hardware. It is mandatory that the Interface Layer sends and receives data from the robot and transforms it in a way that the above layer understands. Ideally, this layer is only one centralized entity that communicates with other(s) outside entity(ies), although, depending on the developer's implementation, it can have more than one if it improves the final application. As seen in the Figure 4.1 after receiving requests from the Skill Layer, the Interface Layer establishes the connection with the outside entity (in this case ROS) to receive the necessary information for its processes.

It is important to mention that each layer communicates with the layer right after and/or before aiming to build a robust architecture. Thus, there is no communication between no adjacent layers, for instance, the bottom layer communicating directly with the first one. In this example, the data have to go through all the layers above the last layer in order to reach the first one.

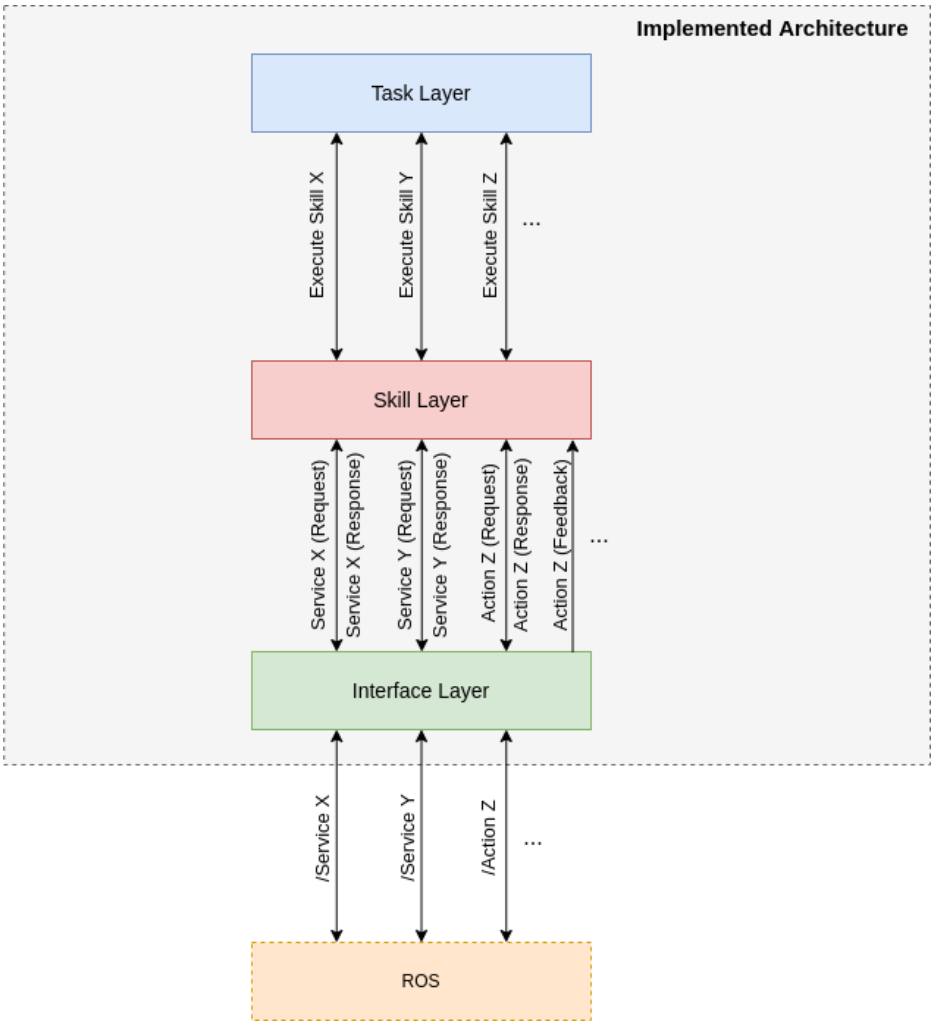


Figure 4.1: Proposed architecture overview diagram.

4.2 Layer Based Approach

In this section, each of the layers in the proposed architecture will be explained alongside implementation details.

As mentioned in the previous chapter, the proposed architecture is divided into three layers:

1. Task Layer;
2. Skill Layer;
3. Interface Layer;

The first and higher level layer is the Task Layer, then below it is the Skill Layer, and the lowest level layer is the Interface Layer.

4.2.1 Task Layer

The Task Layer (higher level entity) is where the main goal of the system is defined, where is designed how the robot accomplishes a goal, that is, the skills to execute, disregarding the implementation details. It describes the tasks which together compose a behavior. They are organized in a tree structure called Behavior Tree(BT), as the following XML code demonstrates(Listing 4.1).

Listing 4.1: Example of a Behavior Tree in XML.

```
<root main_tree_to_execute = "MainTree" >

  <BehaviorTree ID="MainTree">
    <SequenceStar name="approach_pose_sequence">
      <SetBlackboard output_key="approach_pose" value="2.4;0;0" />
      <Fallback name="approach_pose_fallback">
        <Action ID="CheckPose" goal_pose="{approach_pose}" current_pose="{input_pose}" />
        <Action ID="GoToPose" target_pose="{approach_pose}" />
      </Fallback>
    </SequenceStar>
  </BehaviorTree>

</root>
```

In the XML code example(Listing 4.1), the skills are identified by the words "Action ID"and their name. After them, it is possible to declare their inputs and outputs, for example, the *GoToPose* skill have as an input the *approach_pose* declared in the tag *SetBlackboard*.

Finally, each skill described in the BT is triggered in order and executed in the Skill Layer.

4.2.2 Skill Layer

The Skill Layer is where the basic capabilities of a robot (e.g., grasping an object) are designed. It describes the implementation of a BT's leaf node. Leaf nodes can run in mainly two ways regarding the BT engine[77]:

- In the same executable file. Therefore, the source code of the skill is written inside the BT's engine leaf node where the Tick() and the Halt() methods are implemented.
- In a separate executable. Hence, the source code of the skill is written in a separate program that exposes the interface for calling the Tick() and the Halt() methods. A BT's engine leaf node forwards the calls to the corresponding executable.

The approach presented in this dissertation will follow the implementation of the leaf nodes in the same executable file, that is, the Tick() and Halt() methods are in the same executable as the code of the skill. The skills implemented used in the behaviors that will be presented further along are:

- GotoPose
- CheckPose
- CallService
- ApproachObject

4.2.2.1 GoToPose

The *GotoPose* skill has the goal of moving the robot from one point in space to another, as the ability to move, given a goal position, is mandatory in any AMR. It takes the x and y coordinates (linear) and the yaw (angular) coordinate of the goal position as an input and returns *success* in case the robot reaches the goal position or *failure* in case it does not.

The action starts by receiving the goal position (target_pose) from the Task Layer and creating the message of the type *move_base_msgs*, which is the proper message type to communicate with the move base server in the navigation stack(ROS). After that, the message is sent together with the server's name to the Service Layer. This process is executed in the Skill Layer.

To communicate with the move base server, the Interface Layer (Service Node) creates an action client to send the goal position. As it is a ROS action, it provides feedback until it returns the result. Moreover, as explained earlier, if required, the action can be canceled and halted at any time. When the action is finished, the result is sent back to the Skill Layer.

Finally, depending on the result, the Skill Layer report to the Task Layer the node status, which can be success or failure.

It is shown the aforementioned action data flow, the GoToPose BT Node algorithm, and the Service Node algorithm relative to the GoToPose skill in Figures 4.2, 4.3 and 4.4, respectively.

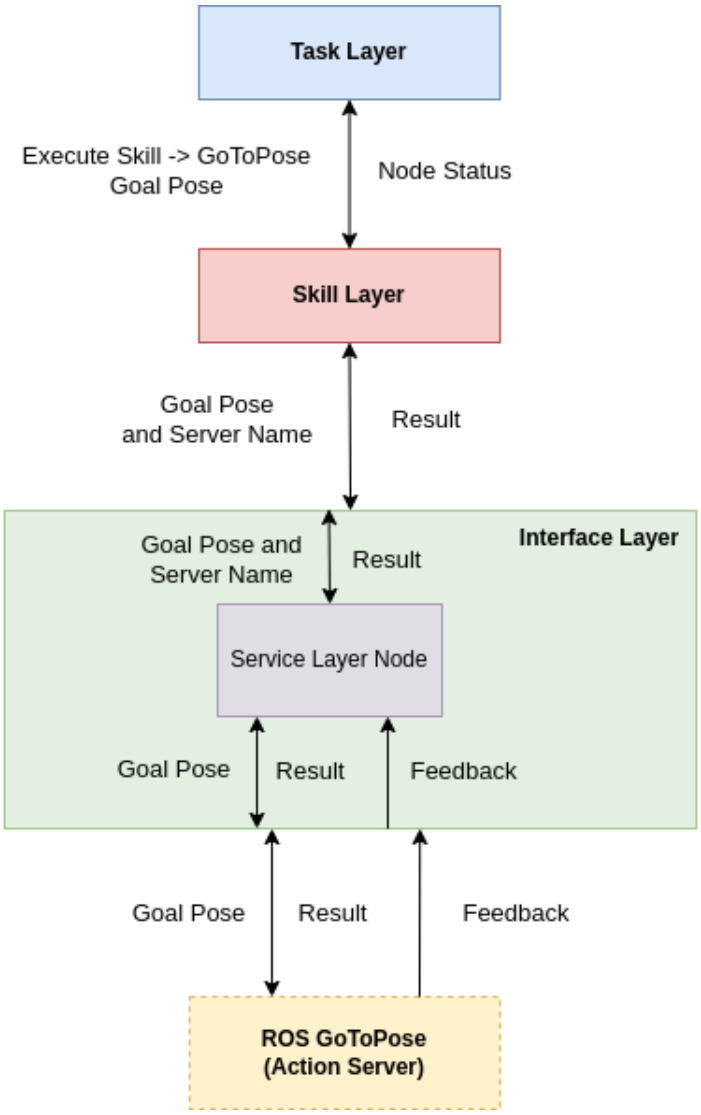


Figure 4.2: GoToPose skill data flow scheme.

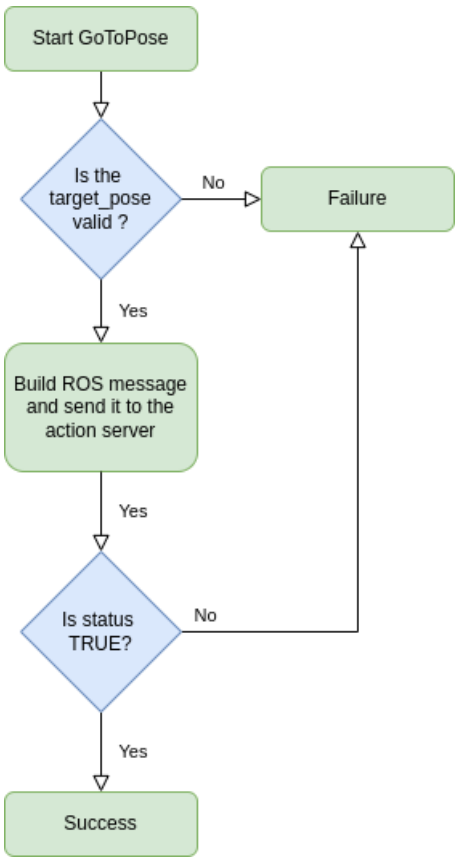


Figure 4.3: BT GoToPose node algorithm flowchart.

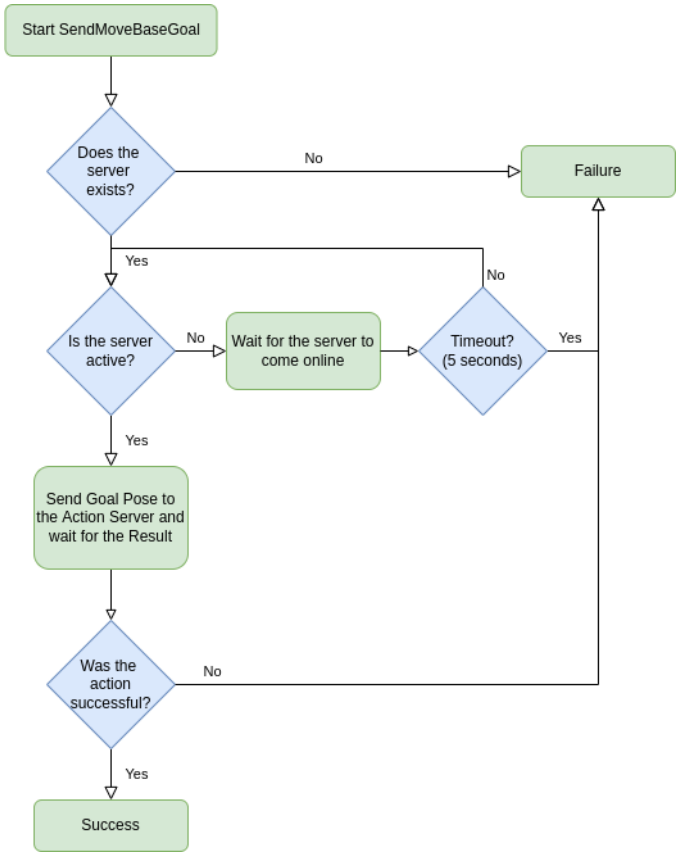


Figure 4.4: Service node algorithm relative to the GoToPose skill.

4.2.2.2 CheckPose

The goal of the *CheckPose* skill is to verify whether the actual position of the robot is equal to the desired position. Thus, the BT Node receives two inputs, the desired position and the current position of the robot, and compares them. The node returns success if they are equal or failure if they are not.

Checking the robot's pose at any given time is important to start new skills (making sure it is in the right position, for instance, to execute the pick of a piece) or to monitor the robot's position (for diagnostics).

The Figure 4.5 shows the algorithm of the CheckPose skill.

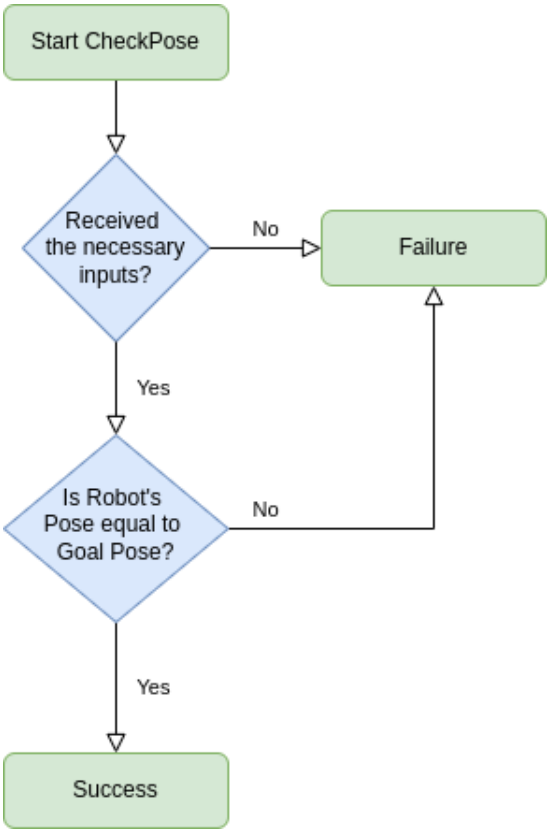


Figure 4.5: BT CheckPose node algorithm flowchart.

4.2.2.3 CallService

The *CallService* skill was implemented with the purpose of triggering any service from the Interface Layer(Service Node). It gets the name of the service as an input and then checks if it exists in the list of services of the Service Node. If it exists, the procedure for making a request to the desired service is executed. If it receives the response message, the CallService BT node returns success to the Task Layer otherwise, it returns failure. Depending on the service, the inputs and outputs of this skill may vary due to the nature of the services. For instance, a service that gets the actual position of the robot may output that position for other nodes. In the Figure 4.6is depicted the flowchart of the CallService skill.

The CallService skill was an implementation choice to generalize the call of services, increasing the system's efficiency. It is completely possible to make service requests without this skill, however, every skill would have to implement a service call procedure to the desired service, which is always the same and is directly attached to the Interface Layer (Service Node). One of the advantages of BTs is increasing modularity and flexibility of systems, therefore creating a skill that allows the detachment of every other skill in the system from the Interface Layer is advantageous. Moreover, the development of generic

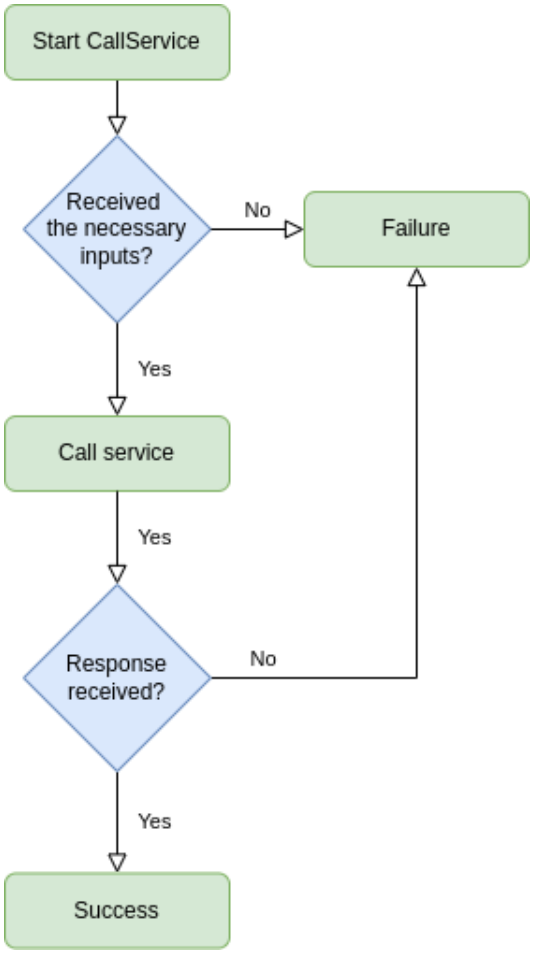


Figure 4.6: BT CallService node algorithm flowchart.

code that can be used for a broad variety of use cases usually increases the pace of development as it doesn't require many changes from case to case. Therefore, the CallService is seen as a valuable skill to the system.

4.2.2.4 ApproachObject

The goal of the *ApproachObject* skill is **to enable the approaching to an object**. As an input, it receives the *object ID*, *object type* and the *object pose* (the object's position and orientation in the environment). It also contains some adjustable parameters such as the dimensions of the object (width and length), the distance desired from it, that is, the distance from the object at which the robot should stop, the error from the goal position allowed for the navigation stack (linear and angular) and the velocity for approaching the object (linear and angular). These parameters are easily adjusted in a separated configuration file (YAML), represented in the Listing 4.2.

Listing 4.2: Approach object configuration file (YAML).

```yaml
chargers:
  - id: 1
    dimensions:
      length : 0.600 # meters
      width : 0.370 # meters
    params:
      distance_from_object : 0.02 # meters
      distance_to_goal_tolerance : 0.1 # meters
      heading_tolerance : 0.01 # meters
      angular_velocity_approach : 0.1 # m/s
      linear_velocity_approach : 0.08 # m/s
      yaw_goal_tolerance : 0.01 # radians

containers:
  - id: 31
    dimensions:
      length : 1.0
      width : 1.2
    params:
      distance_from_object : 0.05
      distance_to_goal_tolerance : 0.05
      heading_tolerance : 0.01
      angular_velocity_approach : 0.2
      linear_velocity_approach : 0.1
      yaw_goal_tolerance : 0.01
  - id: 32
    dimensions:
      length : 1.2
      width : 1.0
    params:
      distance_from_object : 0.5
      distance_to_goal_tolerance : 1.8
      heading_tolerance : 0.4
      angular_velocity_approach : 0.3
      linear_velocity_approach : 0.1
      yaw_goal_tolerance : 0.05
```

The Listing above illustrates not only the parameters of the objects but also their' IDs and types.

The first word, more to the left side, indicates the type of object to approach. In the system presented, there are one charging station and two containers. Below the word 'chargers' is declared the charging station(s) (can be more than one) in the system, in this case, it's only one with the ID number one. Below the word, 'containers' are declared the containers known by the system. There are two containers with IDs thirty-one and thirty-two, respectively. One rule in the declaration of the objects is that they are spaced by thirty numbers, that is, it's possible to declare thirty chargers (from ID one to thirty) and thirty containers (from ID thirty-one to sixty).

Indentation determines the hierarchy of the parameters. The first are the objects (chargers and containers), the second is the characteristics of the objects (ID and dimensions, parameters), and the third are the entities that specify the characteristics such as inside dimensions, there are length and width, and inside *params* are the distances and velocities determined to approach the object.

At this stage, it is important to clarify that the word 'distance' refers to the distance from the center pose. Consequently, when it's mentioned the distance between the robot and the object, it refers to the difference between their center poses.

The execution of the *ApproachObject* BT node (Skill Layer) starts by checking if the object *ID* and *type* are valid and known. In case they are, the server action client is created, and the pose coordinates of the goal are converted to build the necessary ROS action message(Listing 4.3). Completed the process, the BT node sends the server client name and the completed ROS message to the Interface Layer (Service Layer node). The *Service Layer* node simply sends the message received from the *Skill Layer* whenever the action server is available. In this stage, the message sent triggers the algorithm presented in the ROS *ApproachObject* node.

The algorithm starts by gathering from the robot's odometry, published by the navigation stack to know where the robot is. Then it starts looping through the following steps:

1. With the object's pose received, the parallel and perpendicular distance to the object is computed, as well as the yaw difference between the robot and the object;

2. The yaw difference is reduced by moving the robot angularly to align it with the object (the difference must be between zero and the yaw tolerance);

3. The perpendicular distance (y-axis) is reduced by moving the robot sideways, in the direction of the object;

4. The distance between the robot and the object (x-axis) is reduced by moving the robot forward until it reaches the desired position.

The desired position is determined by taking into account the dimensions of the object thus, the robot does not collide with the object. Hence, the *distance from the object* parameter value in the configuration file refers to the difference between the robot and the object boundaries instead of the center poses.

In each iteration, the distances between the robot and the object on each axis are updated and published in a feedback message (Listing 4.3). The approach object message was declared in the *ApproachObject.action* file which belongs to the *amr_msgs package*. This package contains all the custom services, messages, and actions created for the AMR system. It is also important to emphasize that this algorithm is only valid for robots that can move perpendicularly to a point (y-axis).

Listing 4.3: Approach object action message.

```
#goal definition
geometry_msgs/PoseStamped goal_pose
---
#result definition
bool result
---
#feedback definition
float64 x_distance_to_object
float64 y_distance_to_object
float64 delta_angle
```

Finally, when the algorithm finishes, the result from the action server is returned (*success* or *failure*) to the *Service Layer* node, which sends the result to the BT Node (Skill Layer). The *Skill Layer* then reports its status to the Task Layer. Figures 4.8, 4.7, 4.9, and 4.10, describe the algorithm of the BT node, the algorithm of the ROS ApproachObject node, the algorithm of the Service Layer node, and the data flow of the skill, respectively.

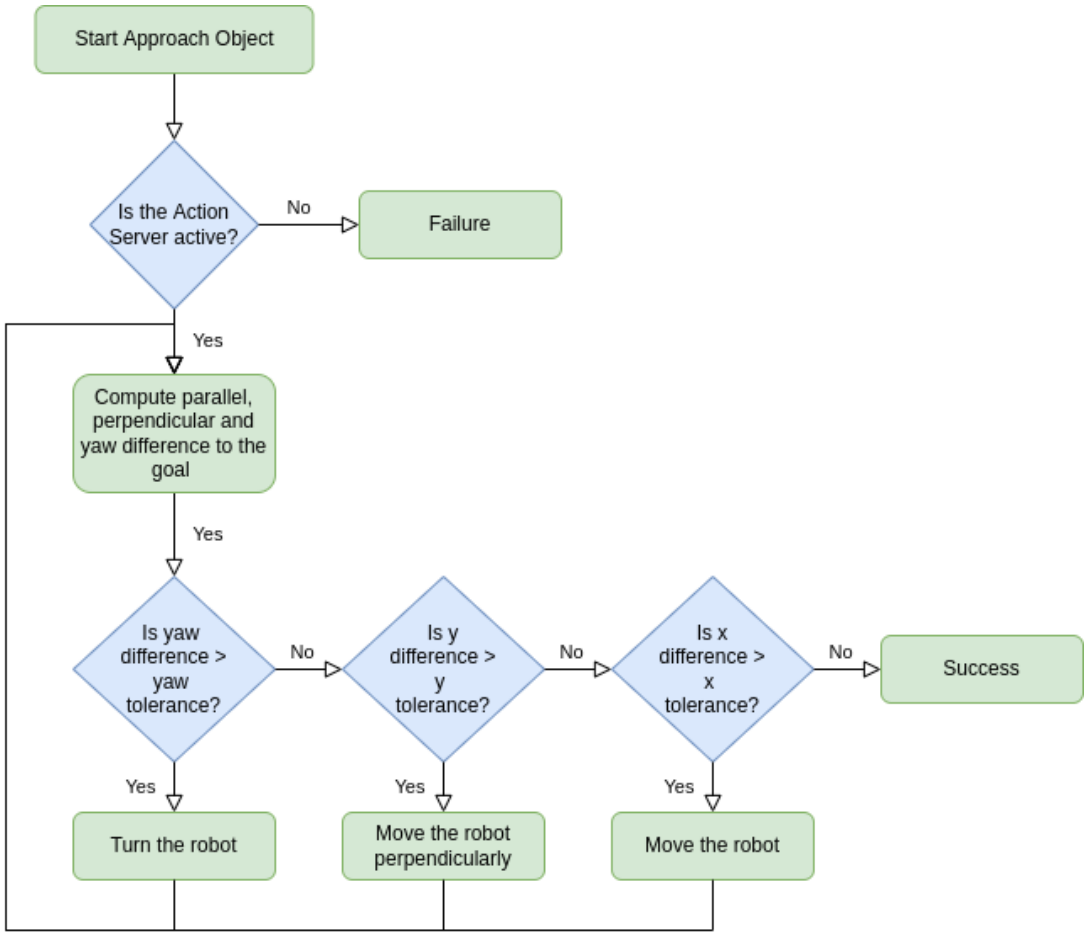


Figure 4.7: ROS Approach Object node algorithm flowchart.

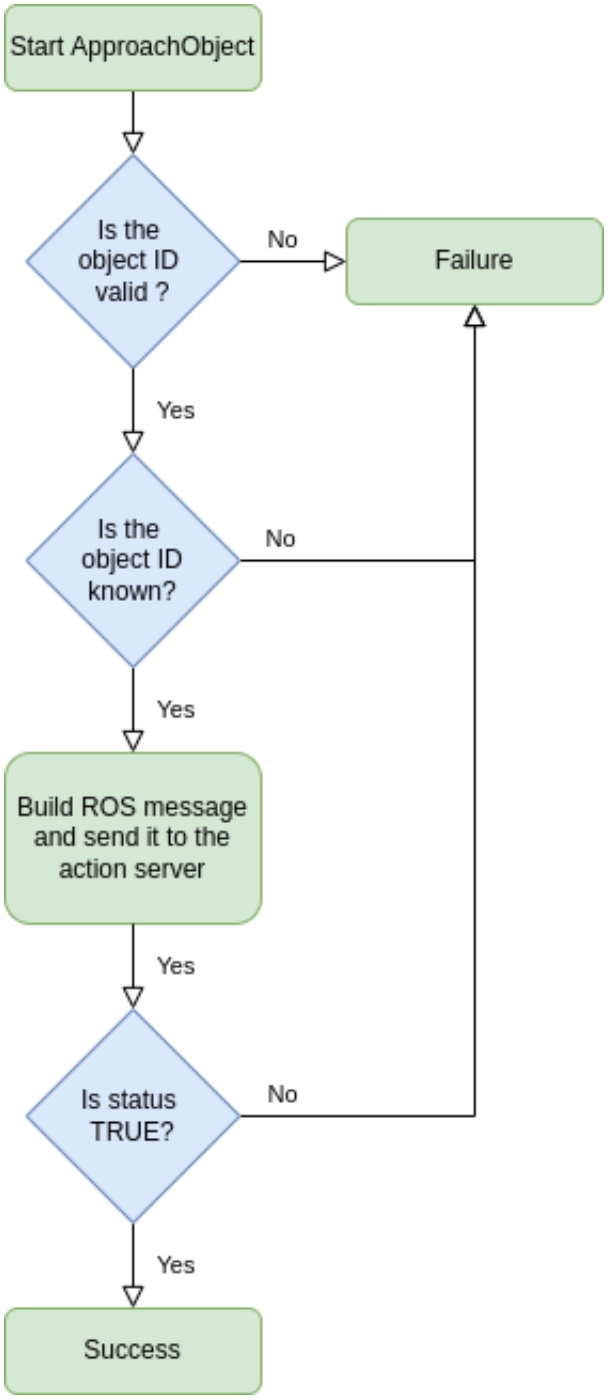


Figure 4.8: BT ApproachObject node algorithm flowchart.

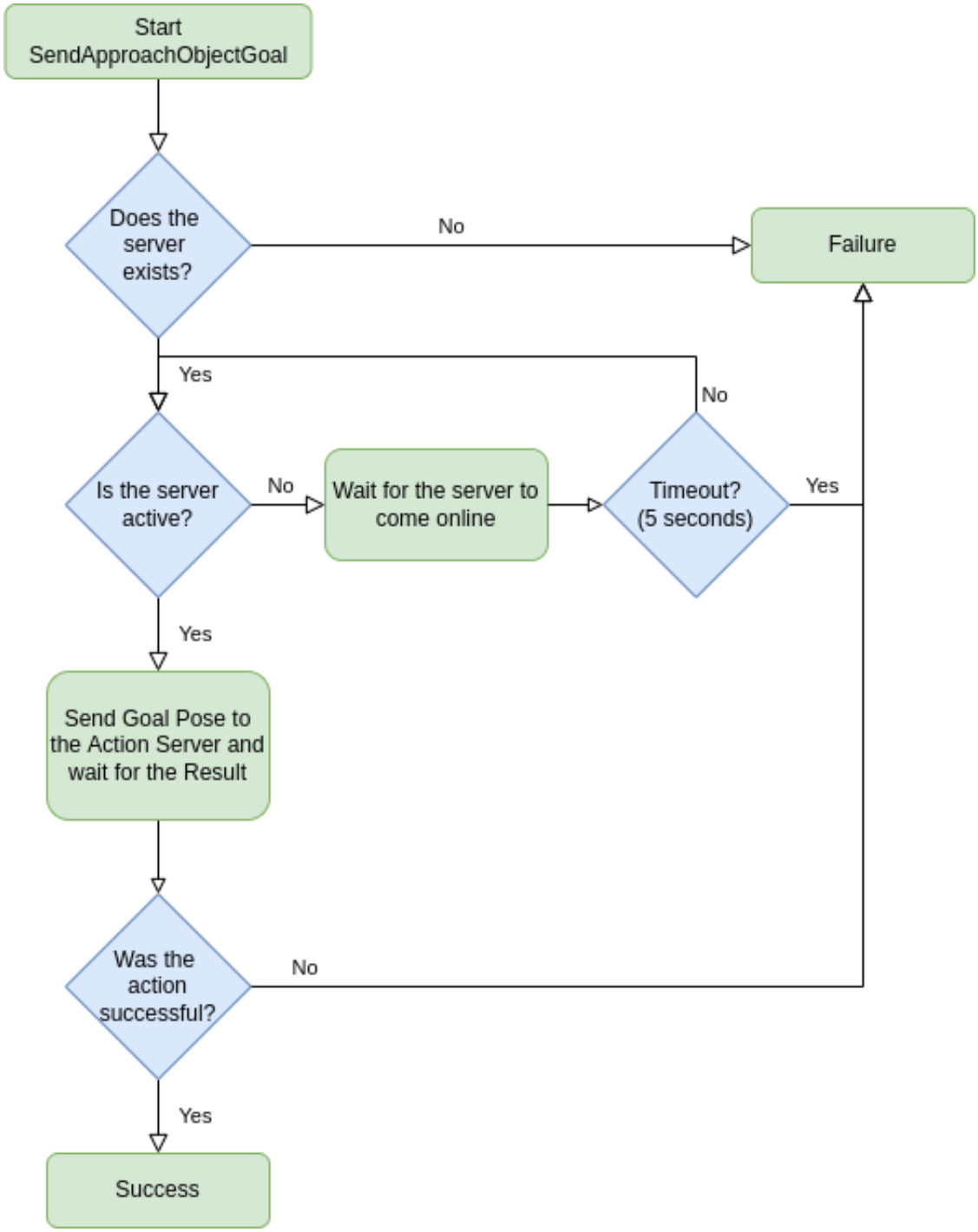


Figure 4.9: Service Layer node algorithm relative to the Approach Object skill.

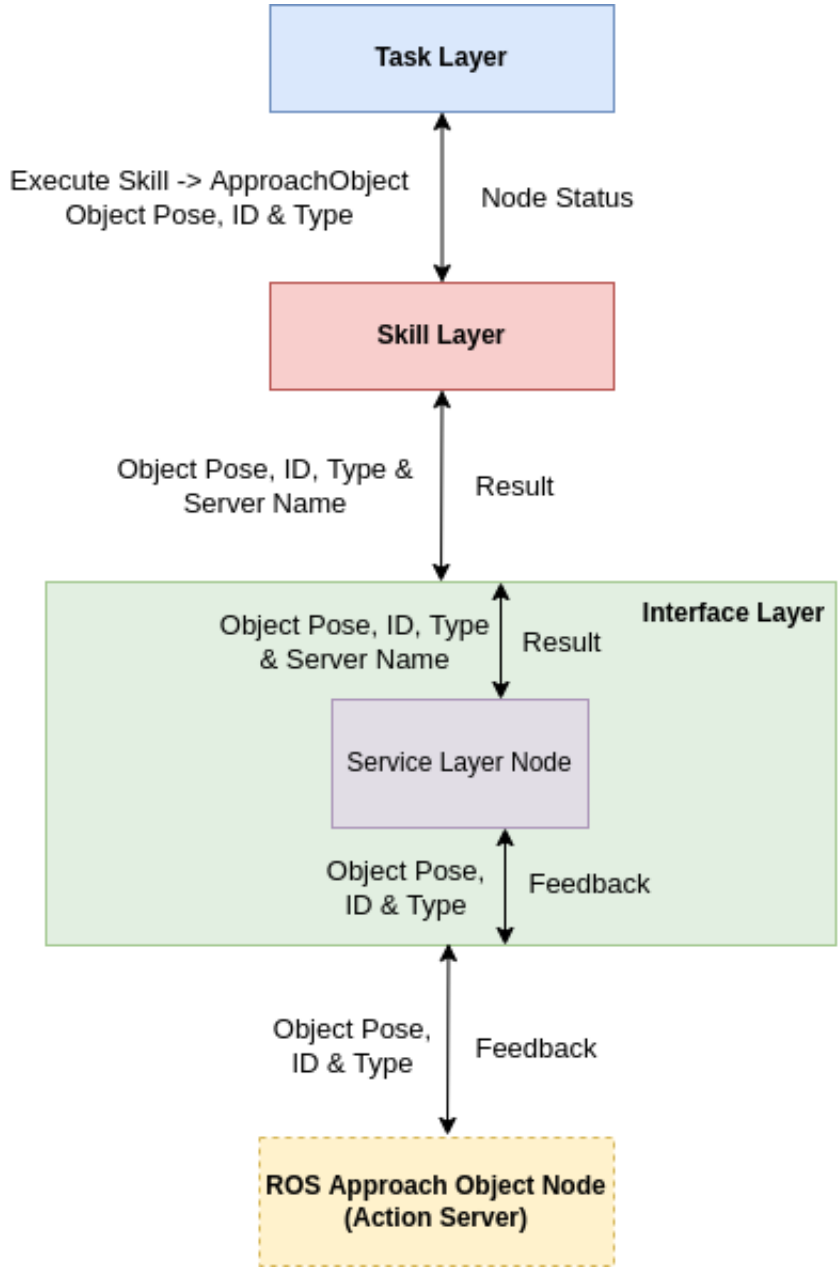


Figure 4.10: BT Approach Object skill data flow.

4.2.3 Interface Layer

The Interface Layer sets a bridge between the implemented architecture and ROS. It provides the connection to ROS services and Actions' servers, so data provided by the sensors and ROS processes can be accessed.

The Interface Layer contains system-level entities that serve as an access point for the skills to command the robot and receive the necessary information (e.g., move mobile base destination, read sensor input). Particularly, this layer is comped by two ROS nodes: the *Service* node and the *ApproachObject* node.

The former possess inner services to communicate with ROS action and service servers in order to send the needed information to the *skill layer* as well as to receive continuously important messages from ROS (e.g., odometry messages which determine the robots' localization).

The services implemented in this node are:

- **Get Robot Pose**. The service *get_robot_pose* gathers the odometry data published by ROS navigation stack and returns the current position of the robot in the environment.
- **Get Object ID**. The service *get_object_id* returns the object ID considering its name as the input value.
- **Get Object Type**. The service *get_object_type* returns an integer representative of a type (zero in case of being a charger and one in case of a container) and receives the object's ID as an input.

The *ApproachObject* node implements the approach algorithm described in Section 4.2.2.4 and provides feedback (Listing 4.3) to the system through the ROS action server.

Therefore these two entities communicate indirectly with each other through the action server.

4.3 Behaviors

As mentioned in the Section 1.3, the definition of robotics behaviors is ample, therefore, a robotic behavior is a set of skills (more than one) organized purposefully to reach a goal. This section presents the behavior implemented for the AMR from the research project CONTIGO[11] in development by INTROSYS[12] and Volkswagen Autoeuropa[13].

Hence, the behaviors explained in detail in the next subsections are the folllowing: *move and check pose*, *approach single container*, *approach multiple containers*, and *charge*.

4.3.1 Move and Check Pose

The ability of the robot to localize itself in the world is crucial to navigate safely and smoothly in the environment. Thus, a repetitive task has to be executed multiple times during navigation to guarantee a good overall performance of robotic system which is checking to inform all the dependent algorithms and robot functions of the robot's pose.

Therefore, three skills where organized as follows creating simple behavior:

1. GotoPose
2. CallService
3. CheckPose

Creating a behavior similar to the *move and check pose* will allow the robotic system to continuously know its location after a change in position. The Figure 4.11 represents the behavior, graphically.

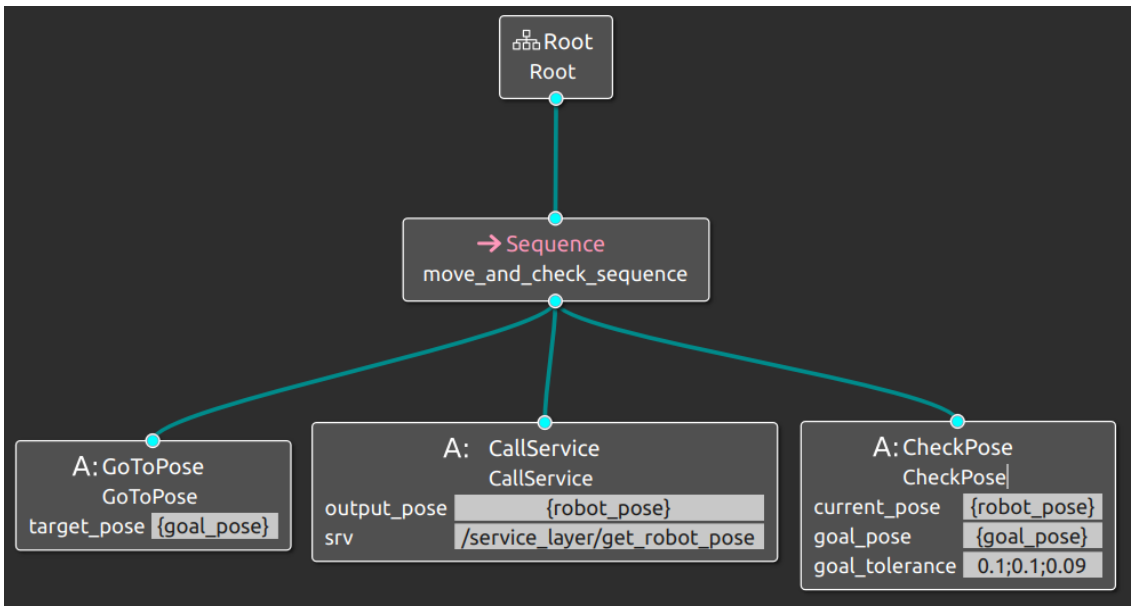


Figure 4.11: Move and check pose behavior tree.

The tree represented above is composed of a sequence of the three skills mentioned earlier. From a known initial position the robot moves to the *goal pose* received as an input, then the *CallService* skill receives as an output the current pose of the robot. Finally, the robot's pose is compared with the goal pose in the skill *CheckPose*. The result of the last skill, being positive if the robot is in the goal position or negative if it isn't, can then be used to trigger other action from another BT according to the system needs.

4.3.2 Approach Single Container

The *approach single container* behavior represents a mandatory task performed by the robotic system in the project CONTIGO. The robot have to approach a container to pick the pieces necessary to deliver to the operator. Therefore, it was created a generic node which gives the robot the ability to approach any object.

The behavior is composed of two skills:

1. CallService
2. ApproachObject

The Figure 4.12 illustrates the *approach single container* behavior.

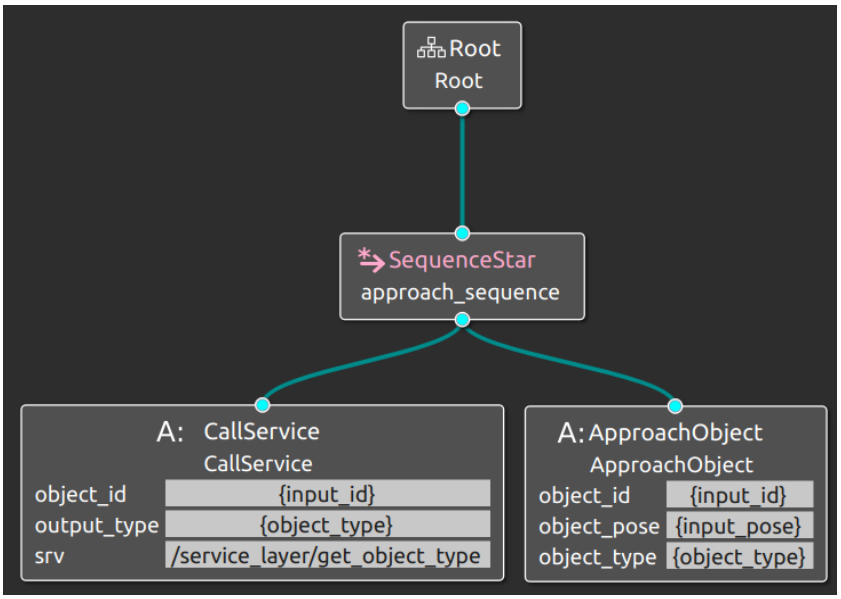


Figure 4.12: Approach object behavior tree.

Firstly, the *CallService skill* receives the object type to be approached (in this case, the container), which is needed to execute the next skill *ApproachObject*. Then, the algorithm to approach the object is triggered receiving as an input the *object_id*, the *object_pose* and the *object_type* sent by the previous skill.

4.3.3 Approach Multiple Containers

Although approaching a single object is useful, the robotic system have to be capable of scalĩng the task, that is, approaching multiple containers with different pieces necessary to be picked and transported. Therefore, the *approach multiple containers* behavior was created based on the previously explained one. The Figure 4.13 shows the *approach multiple containers* behavior.

The tree is composed of two sequences: the first is a high level sequence which coordinates the main skills to be executed; the second creates a final behavior after three of the four containers have been approached.

In the figure each *ApproachContainerTree* is a sub BT which has another tree inside. This hierarchization of trees represents one of the advantages of BTs. In this way it's easier to read and understand the system's behavior.

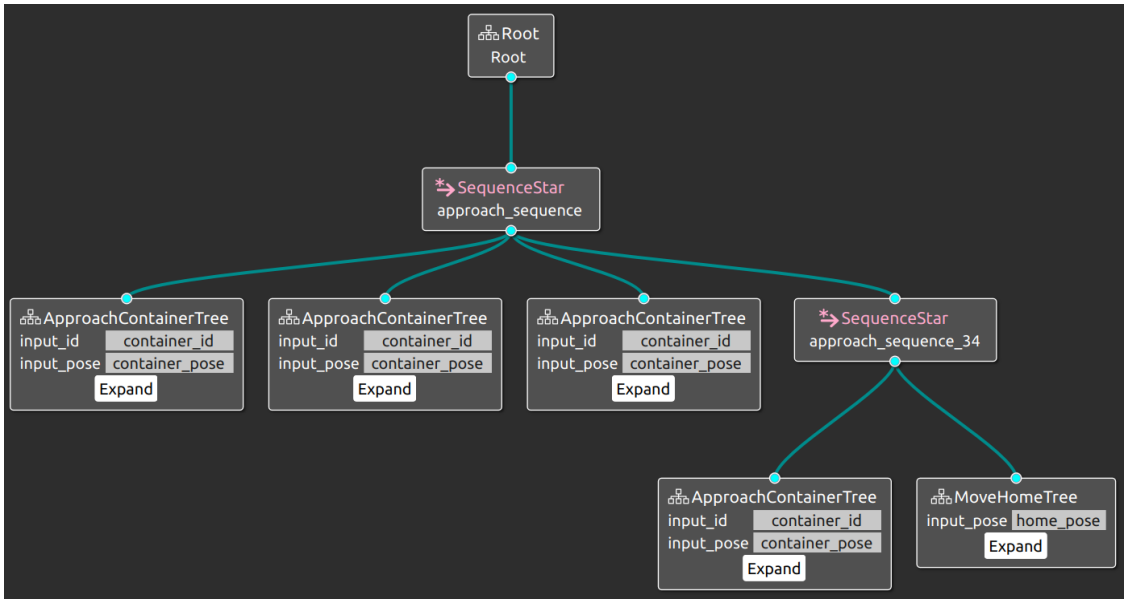


Figure 4.13: Approach multiple objects behavior tree.

Each *ApproachContainerTree* has an *approach single container* behavior inside and the only thing that changes is the *container_id* and the *container_pose*. The final sequence ends with the robot moving to the home position executed by *GotoPose* skill.

4.3.4 Charge

Autonomous battery charging should be a crucial task for future intelligent mobile systems which run on batteries. At any time, it is critical for the system to be aware of the amount of battery left and have procedures in case it's necessary to charge, otherwise, it might result in some undesirable, even catastrophic consequences. Regarding factories' shopfloor, it is even more important this kind of awareness as critical and dangerous processes might be in execution. The work herein presented thus offers a solution by defining a **Charge** behavior.

To accomplish autonomous charging, a set of tasks need to be performed. Thus, the aforementioned skills in the Subsection 4.2.2 were organized to create the **Charge** behavior.

The order of execution is the following:

1. CallService
2. CheckPose
3. GoToPose
4. ApproachObject

The Figure 4.14 shows the Behavior Tree (BT) of the *Charge* behavior, in a horizontal layout, using the BT visualization tool *groot* [78] and, the Listings A.1 and A.2 presented in the appendix A of this dissertation, describes the same BT and its *subTrees*, in XML code format. The full tree with the expanded subtrees is presented in Figure B.2 inside appendix B.

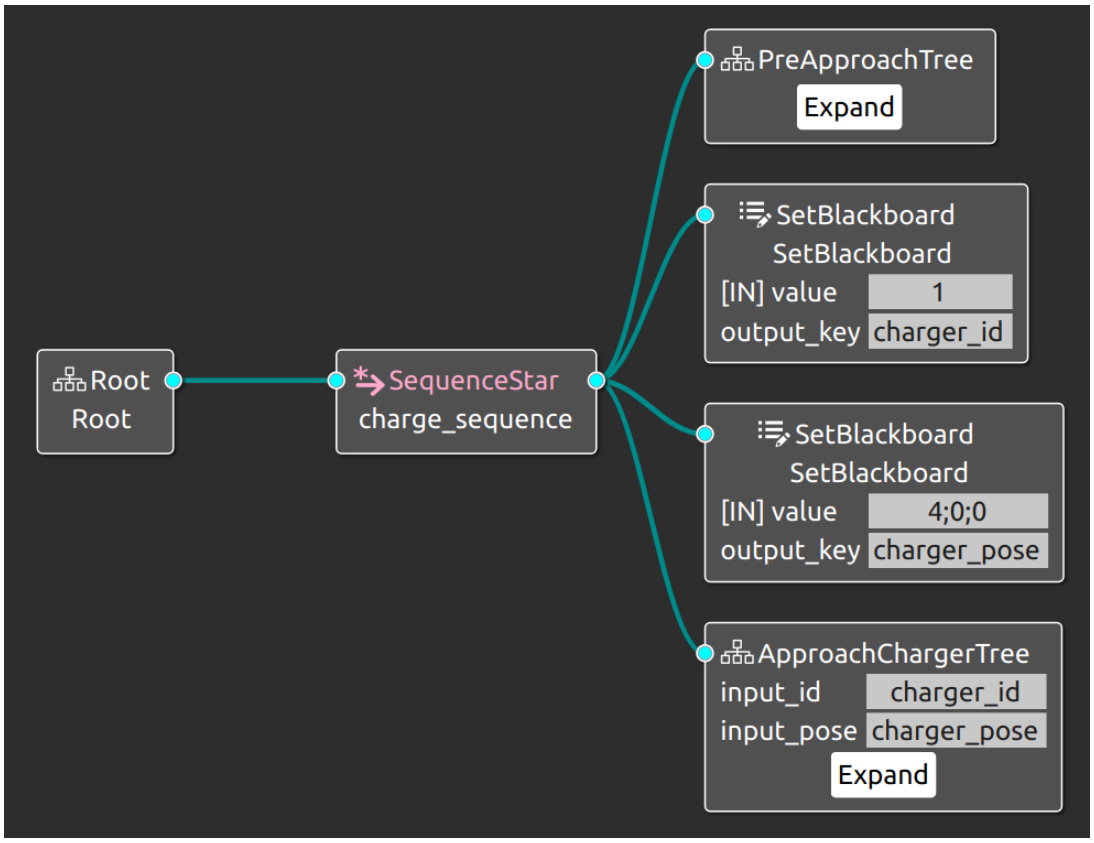


Figure 4.14: Main Tree of the charging behavior.

The main tree is composed by a sequence of instructions. Firstly, it is a sequence node because the goal is to execute a set of dependent instructions, hence if one of them fails, the whole process fails as well, instead of, for example, a *fallback* type node which *tick*[1] the next child with the previous one reporting *failure*.

Secondly, as shown in the Figure 4.14, the main sequence has the type *sequence star*, since the desired behavior is to *tick*[1] the child the node which previously returned *failure*, thus the whole sequence is not restarted, and the procedure does not repeat from the beginning as it would if it was a normal sequence node.

The main BT starts with the execution of a *subtree* called *PreApproachTree*. *SubTrees* are Behavior Trees inside other trees which are hierarchically at a higher level and used with the similar purpose of functions in code. That is, to clean the main tree improving its readability, and provide more modularity, flexibility, and reusability throughout the whole tree structure. For example, the *PreApproachTree* is a *subtree* of the main tree,

although it can have one or more trees inside (*subtrees*) to describe inner actions.

The Figure 4.15 shows the *PreApproachTree*.

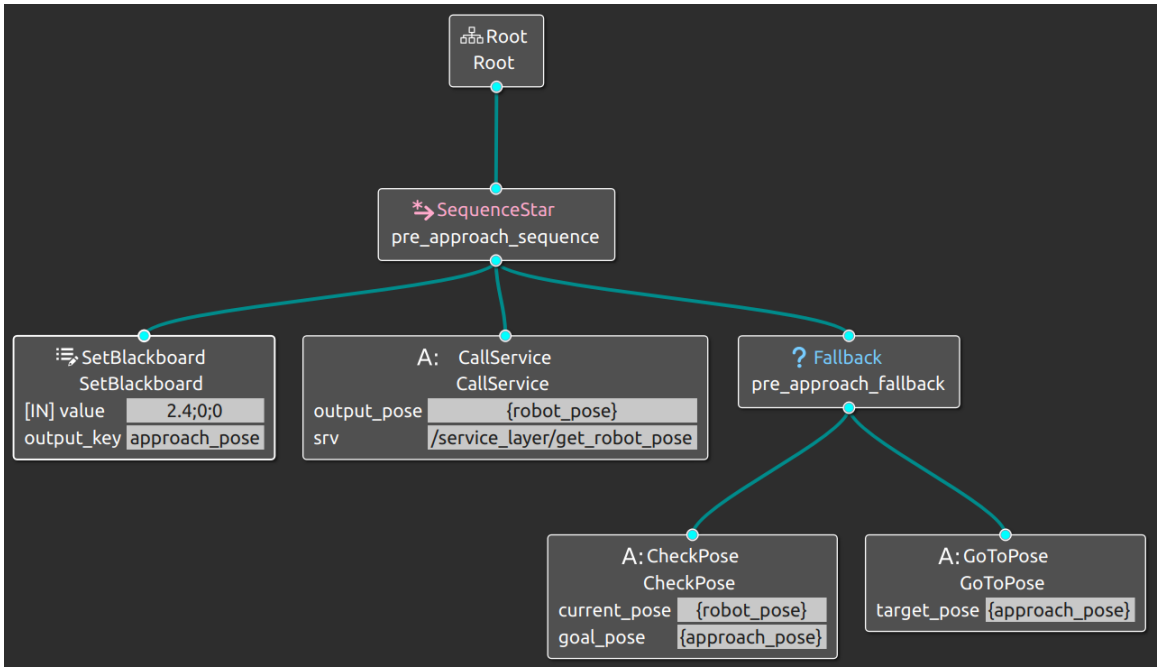


Figure 4.15: SubTree *PreApproachTree* of the charging behavior.

For the same reasons aforementioned earlier, the main sequence has the type *sequence star*.

The main sequence starts by declaring the variable *approach_pose* with the action *SetBlackboard*. The value of the variable has the type *Pose2D*, a created custom type that describes three coordinates: the position on the x-axis (first number), the position in the y-axis (second number), and the rotation in the x-axis (third number). Therefore, the *approach_pose* variable describes the position: $x = 2.4$, $y = 0$ and $yaw = 0$.

Then, the *CallService* action node (represented in the Figure 4.15 by the 'A:') with the service input *get_robot_pose* (*srv* field) to get the current robot's pose. This information is important for the system to be aware of the robot's localization, thus, the trajectory to the closest charging station can be planned and, for example, in case the battery is not enough to reach the station, it can output an alert to the operators or the people around the robot signing a critical situation.

The output of the *CallService* node, the current robot's pose (*robot_pose*) is used as an input of the *Fallback* node (pre_approach_fallback). This node type only *tick*[1] the next node if the current had returned *failure*. Therefore, the two poses (the current robot's pose and the approach pose) are compared by the action *CheckPose*. As the robot's initial pose is at the map's frame center, the next action *GoToPose* is triggered, commanding the robot to move to the desired approach pose.

Returning to the main tree, the variables *charger_id* and *charger_pose* are set to be used as an input of the subtree *ApproachChargerTree* (Figure 4.16).

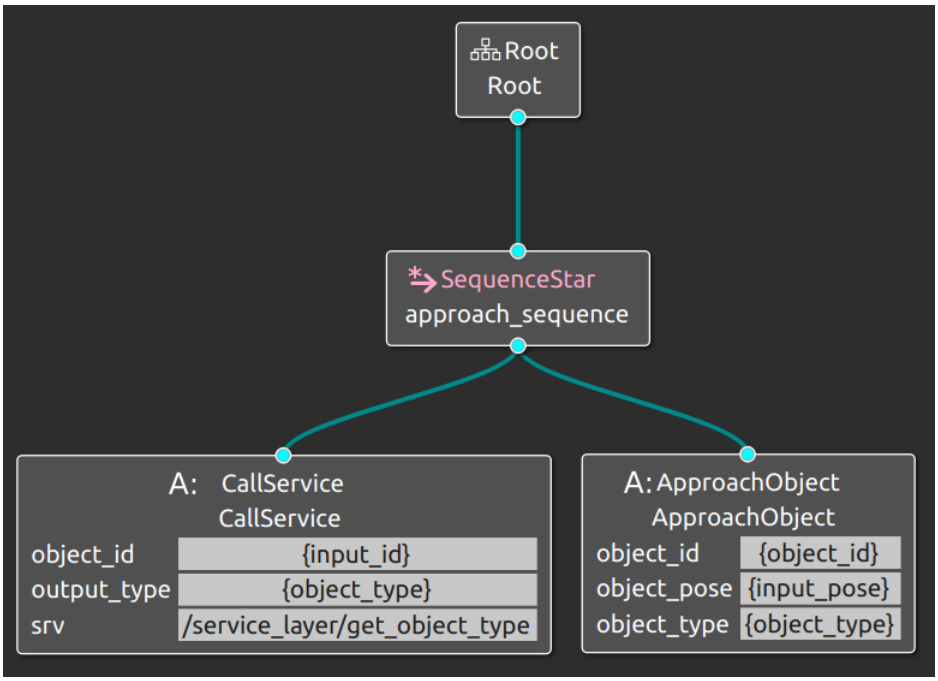


Figure 4.16: SubTree *ApproachChargerTree* of the charging behavior.

The *ApproachChargerTree* is composed by two skills (action nodes): the *CallService* and the *ApproachObject*. The former is necessary to get the object type (service *get_object_type*) needed for the next node. The latter receive as inputs the *object_id*, the *object_pose* remapped as *input_pose* when being used by the subtree *ApproachChagerTree* (Figure 4.14), and the *object_type* output of the action *CallService*.

To conclude, the *Charge* behavior can be divided into two phases: the pre-approach phase, where the position of the robot is acknowledged by the system, and it moves to the desired position prior to the execution of the approach algorithm; and the approach phase where the necessary inputs are reunited to successfully perform the charging station's dock procedure.

In the next sections, all of these procedures are validated in a simulated environment to conclude the real advantages of this type of architecture in the development of robotic systems.

5

Experimental Results and Discussion

This chapter aims to present the results obtained during the evaluation of the architecture and behaviors implemented when facing a recreation of a real-world scenario.

Testing was performed with increasing complexity levels, from the simplest to the more challenging ones. Every test case was performed in a controlled and simulated environment. Nonetheless, simulated test scenarios can give a good approximation of how the system would respond to certain types of environmental changes, and a good comparison can be made between the theoretical and experimental aspects.

Moreover, the layer-based architecture is tested more in-depth with multiple case scenarios. Advantages and possible drawbacks due to the implementation of the system using BTs and the layer-based approach are identified, and a general reflection is made about the approach followed.

5.1 Simulation Overview

The main simulation engine used for the development of this dissertation was *Gazebo* [79], the default ROS simulation environment. Although the robot's model and the other components of the simulation can be visualized in *Gazebo*, *RVIZ* 3D visualization tool ROS' plugin [80] was adopted as *RVIZ* is less computationally demanding compared to *Gazebo* and it also provides more important visualization resources from the navigation perspective, for example, costmap layers, transform frames and planning paths. Therefore, the main demonstration environment for all the test cases, presented in the next sections, is *RVIZ*.

The main models used in the simulation are:

- The AMR (Figure 5.1) with the box for holding the car components, the robotic arm on the top and the magnetic gripper in its flange, and, next to it, the structure for holding the car parts while the rotation and the QR code verification processes are being executed.

- The charger (Figure 5.2) with six spring-like charging pins (meaning they can contract about 2cm to the inside) where the AMR dock and the contact with the metal plates start the charging procedure.
- The environment (Figure 5.3) which is consisted of four delimiting walls in a square-shaped form.
- The container (Figure 5.4) where the pieces are to be grasped by the robotic arm (however, the grasp of the pieces is out of the scope of this dissertation).

For navigation the ROS planners used are the *NavFn*[37] as a global planner and *TEB*[81] as a local planner.

Furthermore, a tool called *Groot*[78] is used to visualize the structure of the Behavior Trees created in XML code format.

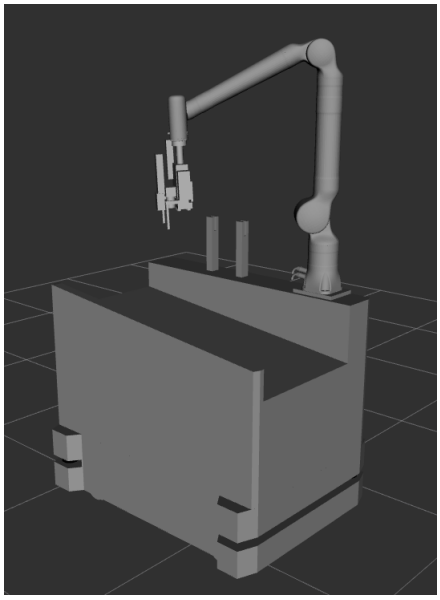


Figure 5.1: Model of the AMR in RVIZ. The sloped is where the boxes for the pieces stand. The robotic arm is on the top of the structure with the magnetic gripper on its flange. The two pieces near the arm are the supports to hold the grasped objects.

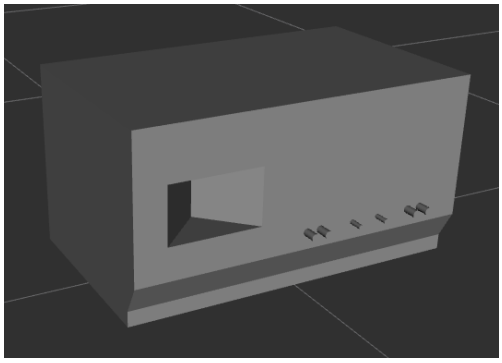


Figure 5.2: AMR's charger. Concave triangle to provide a model matching reference to laser scans and the six charging pins (two for the positive terminal, two for the negative terminal, one for the CAN line negative reference, and one for the CAN line positive reference).

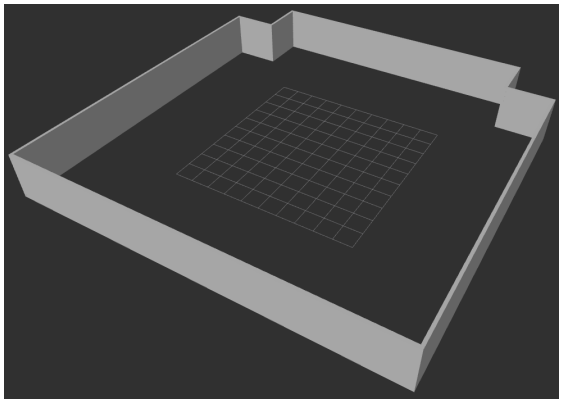


Figure 5.3: Model of the square-like shape environment in RVIZ.

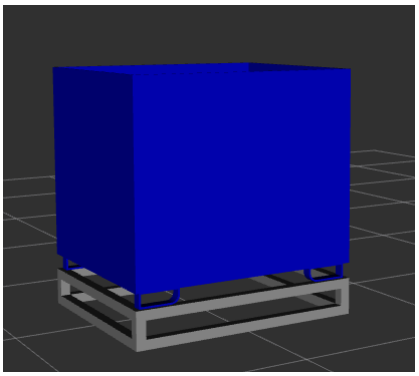


Figure 5.4: Model of the container in RVIZ.

5.2 Hardware Overview

In this section, the hardware of the system in which real-world tests will be performed is presented.

Firstly, the Autonomous Mobile Robot (Figures 5.5 and 5.6). In the figure below (Figure 5.5), it is displayed the AMR's base structure. Hardware details cannot be disclosed since it is still a prototype, hence it is INTROSYS's proprietorship. Nonetheless, the robot can be described more generally. It carries six cameras, four on the sides, one on the back, and another on the front, for full 360-degree 3D image perception, two laser scans for 360-degree coverage, four mecanum omnidirectional wheels powered by two computers, a dedicated processing unit (Nvidia Jetson AGX Xavier) and a fully centralized power management system. Each motor of the AMR has a 0.51 KW rated power, which leads to a total power of 2 kW and 2.7 hp. The computers and the dedicated GPU's characteristics are the following:

- **Computer 1:**
  - Intel Core i7-9700TE CPU;
  - Intel UHD Graphics 610 GPU;
  - 16 GB of RAM;
  - SSD 256 GB of storage.
- **Computer 2:**
  - Intel Core i7-8665UE, Quad Core, 8M Cache, 1.7 GHz CPU;
  - Intel UHD Graphics 610 GPU;
  - 8 GB of RAM;

  - SSD 256 GB of storage.

- **NVidea Jetson AGX Xavier**
  - 512-Core Volta GPU with Tensor Cores;
  - 8-Core ARM v8.2 64-Bit CPU;
  - 32 GB of RAM;
  - 32 GB eMMC storage;
  - MicroSDXC Sandisk Extreme Pro 256 GB of storage;

Figure 5.5: The base structure of the AMR.

In the Figure, is depicted the top structure of the AMR. In the center is the place for the 12-box compartment for the car parts. On the top of the structure is attached the robotic arm with the magnetic gripper on its flange, and on the side of the robotic arm is a structure to hold the piece while the process of rotation and QR code verification is being executed.

The robot's two parts are separated due to the development of the perception/navigation and the manipulation being executed in parallel.

Regarding other important components in Figures 5.7 and 5.8, the robot charger and the container where the pieces remain are exhibited, respectively.

Figure 5.6: Top structure of the AMR.

Figure 5.7: The charger of the AMR.

Figure 5.8: The container from where the car components are picked.

Finally, the laptop used to run all the simulation processes is a Lenovo ThinkPad E580 with the following characteristics:

- CPU - Intel Core i7-8550U, 1.80Hz (8 cores);
- Graphics Card - Radeon 550X and Intel UHD Graphics 620;
- Memory - 16GB of RAM;
- Storage - SSD 250GB.

5.3 Simulation Test Case: Go to Pose and Check Pose

The first test case is the most simple one, and it has the **goal to demonstrate the ROS navigation between two points using BTs**. As aforementioned earlier, it is achieved through the communication between the Interface Layer and the ROS Move Base server.

Therefore, the BT built for this example is composed by three skills: *GoToPose*, *CheckPose* and *CallService*.

Firstly, the AMR receives a pose to move to, using the *GoToPose* skill. Then, when it arrives at the goal, the *CallService* skill is triggered to get the robot's pose. Finally, the outputted pose from the *CallService* skill (the robot's pose) is compared with the goal pose by the BT node *CheckPose*. The Figures 5.9, 5.10 and 5.11 shows the BT of this test case and the respective skills being executed (highlighted in green). The highlight in green means that they were successfully completed, otherwise the highlight color would be red. Additionally, the Listing A.3 represents the XML code of the BT.

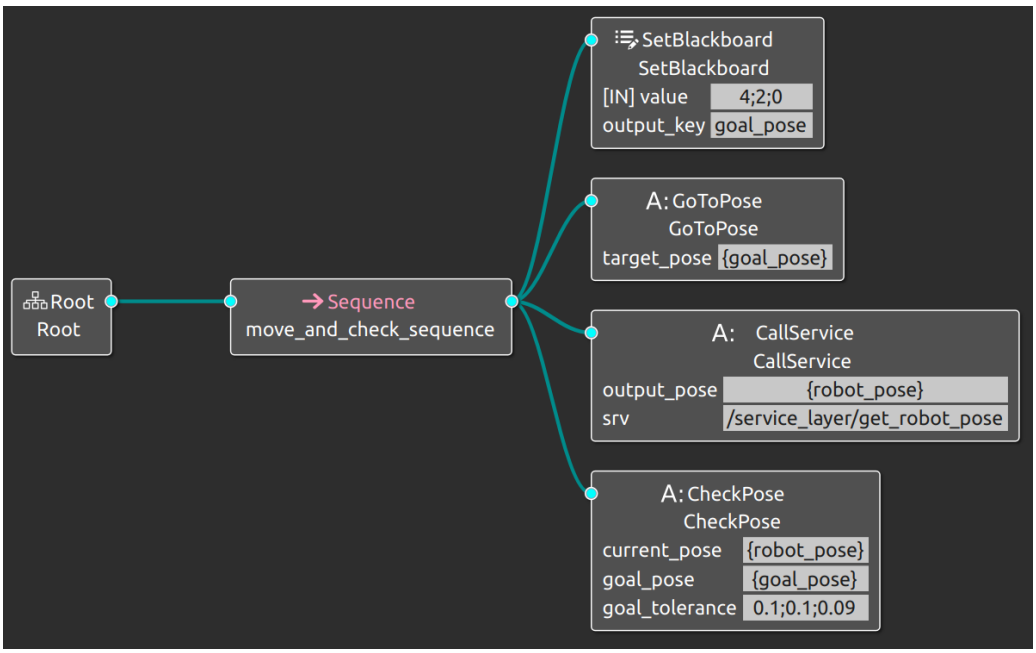


Figure 5.9: Structure of the test's case 5.3 Behavior Tree.

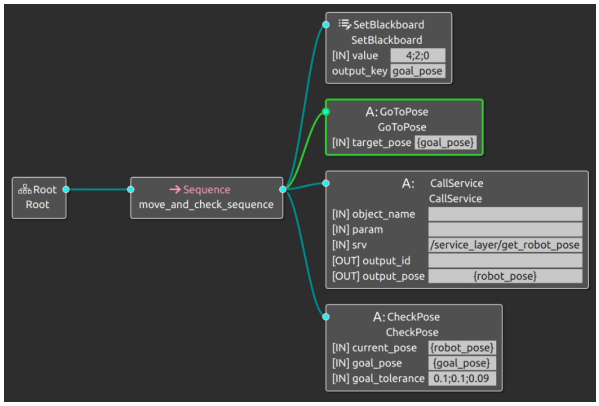


Figure 5.10: Visualization of BT nodes *GoToPose* execution success (highlighted in green).

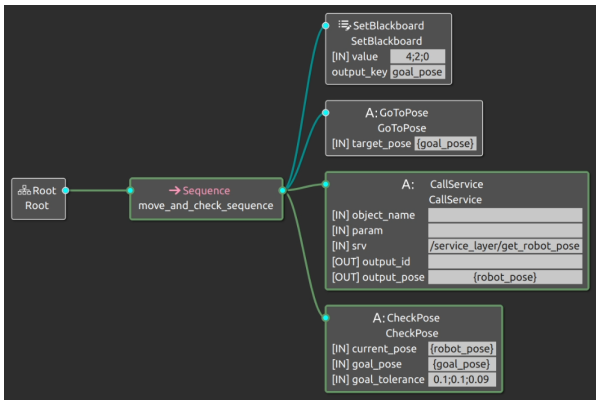


Figure 5.11: Visualization of BT nodes *CallService* and *CheckPose* execution success (highlighted in green).

In the Figure 5.12, the execution of the skill *GoToPose* can be visualized, in simulation, by the AMR's trajectory (green line). The full process is displayed in the Figure B.1 in appendix B and a video[1] is available to better perceive the behavior of the AMR.

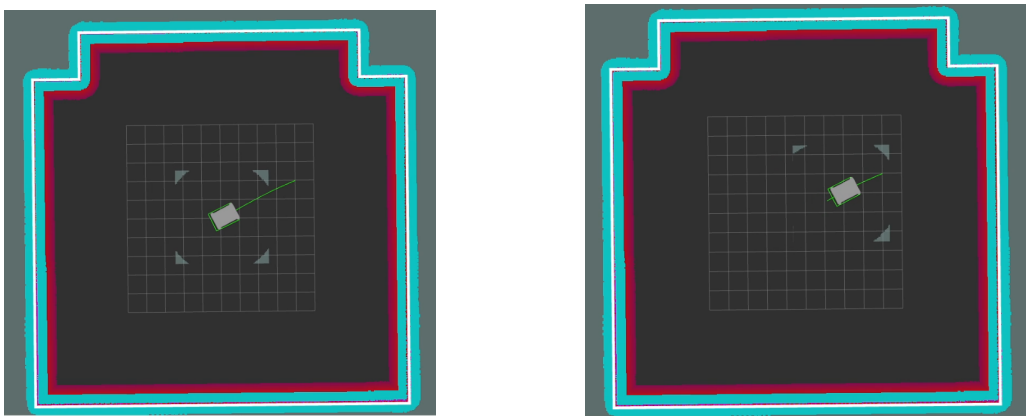


Figure 5.12: Trajectory took by the AMR in the sequence (green line).

[1] https://youtu.be/3TLEJiWs2CI

5.4 Simulation Test Case: Move and Check Pose (failure)

The **goal** of this test case is **to validate the *CheckPose* BT node**. The skills selected are the same compared to the test5.3, however, the structure of the tree is different (Figure 5.13 and XML code in Listing A.4 of appendix A).

There are two sequences nested in a fallback node, thus, if the first sequence returns *failure*, the next sequence will be *ticked*[1]. As the *CheckPose* node is meant to be tested, the first sequence goal pose is not the right one on purpose. Therefore, the BT node will return *failure* and the second sequence is triggered. Since the robot went to the correct goal pose, the second sequence will return *success*.

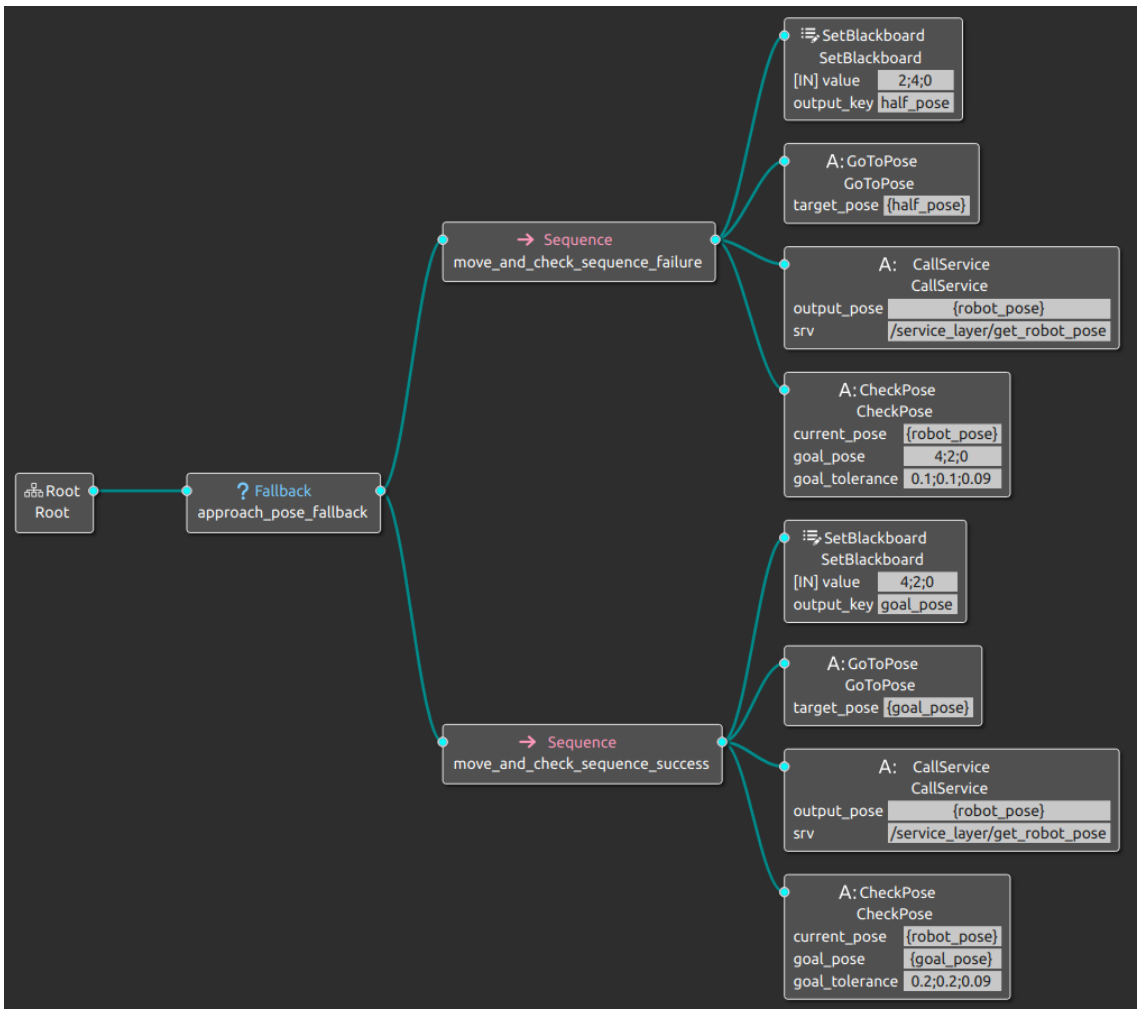


Figure 5.13: Structure of the test's case 5.4 Behavior Tree.

In the Figure 5.14 is shown the BT's state when the first sequence fails, due to the BT node, *CheckPose*, failure (highlighted in red). This means that the robot pose is not in the correct pose, thus, the second sequence is executed (highlighted in orange). Finally, the second sequence returns *success* because the robot is in the correct pose (Figure 5.15).

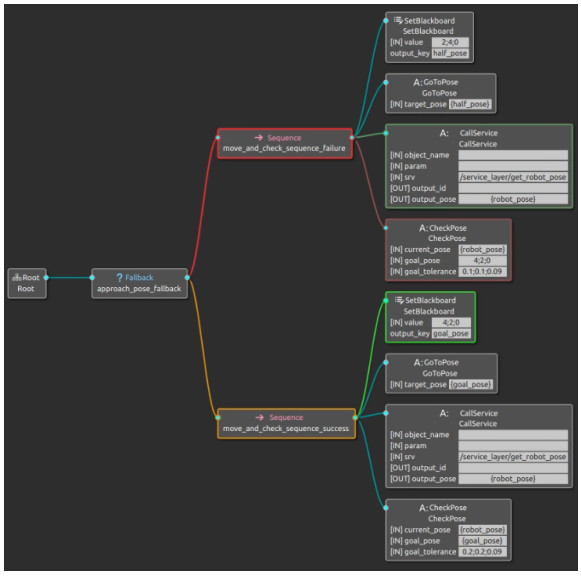


Figure 5.14: Behavior Tree of the test case 5.4 in execution. The first sequence returned failure (highlighted in red), and the second sequence is being executed (highlighted in orange).

Regarding the simulation, the execution of the trajectories on the first sequence (Figure 5.16) and on the second (Figures 5.17, 5.18 and 5.19, respectively) is represented by the green line. Furthermore, a video[2] is available to better understand the simulated behavior of the AMR.

5.5 Simulation Test Case: Approach to the Charging Station

This test case has the **intention of validating the *Approach Object* BT node**. The AMR's charger will be added to the environment, and its coordinates are used as the input to the BT node. Therefore, the AMR should align with the charger regarding all the axis and stop in front of it, according to the distance parameterized.

The AMR starts with a pose that differs from the charger in both linear (x, y, and z) and rotational axes (row, pitch, yaw), thus, the alignment can be perceived.

As explained in the Section 4.2.2.4, the *Approach Object* BT node starts by aligning the orientation of the AMR with the orientation of the object (charger). The Figure 5.20 shows the initial pose of the robot, whereas the Figure 5.21 displays the orientation alignment process.

[2] https://youtu.be/bdvpZt3gZ6I

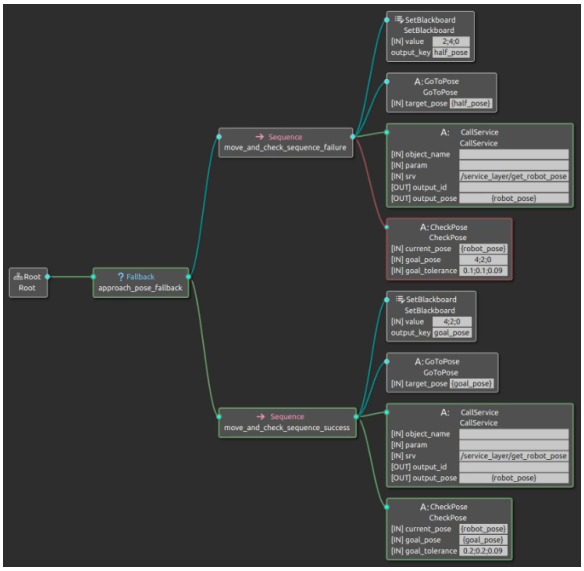


Figure 5.15: Final state of the test case 5.4. The second sequence returned success since the position of the AMR is the predefined goal pose.

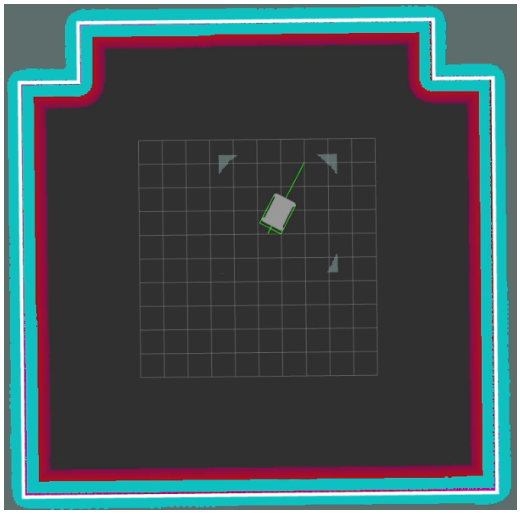


Figure 5.16: Trajectory (green line) taken by the AMR on the first sequence.

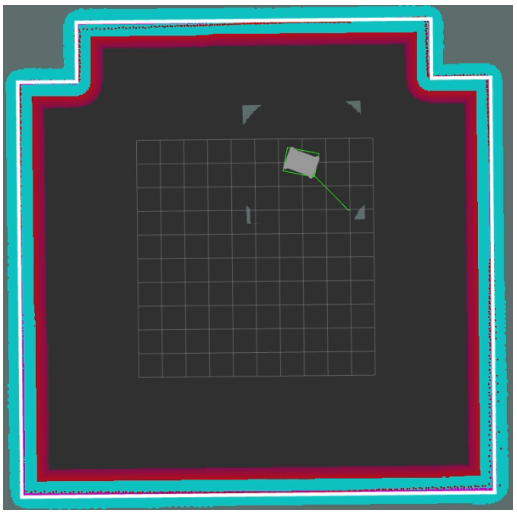


Figure 5.17: Trajectory (green line) taken by the AMR on the second sequence.

Secondly, the next step assures that the AMR is aligned on the y-axis with the charger, which can be seen in the Figure 5.22.

Finally, the AMR decreases its distance from the object until it reaches the desired distance (Figure 5.23).

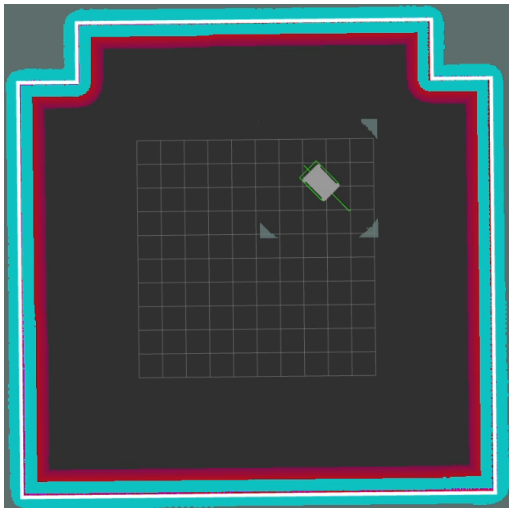


Figure 5.18: Trajectory (green line) taken by the AMR on the second sequence.

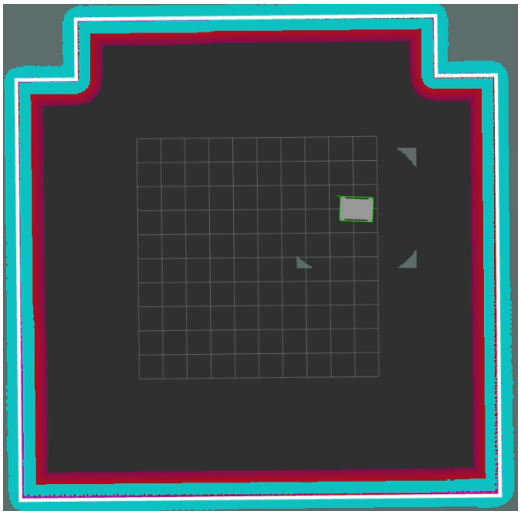


Figure 5.19: Final pose of the AMR, where the *Check Pose* BT node is being executed for the second time.

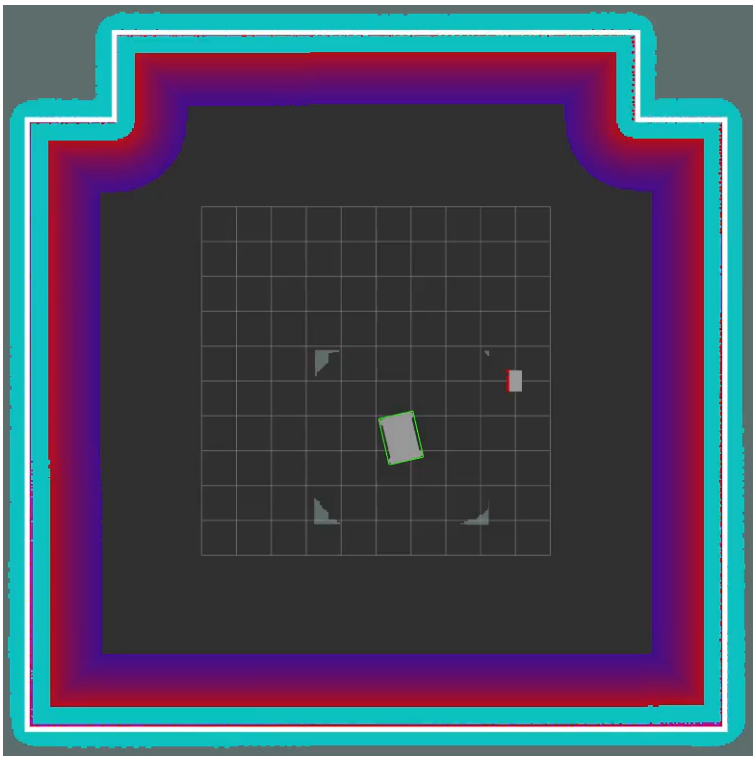


Figure 5.20: AMR's initial pose in the Approach Charging Station test case.

A 3D view (Figure B.12 in appendix B) and a video[3] are available to better visualize the test performed.

According to the B.13 and the know variables - the length of the robot (1.3234m) and

[3] https://youtu.be/Sj8151Cd9ZQ

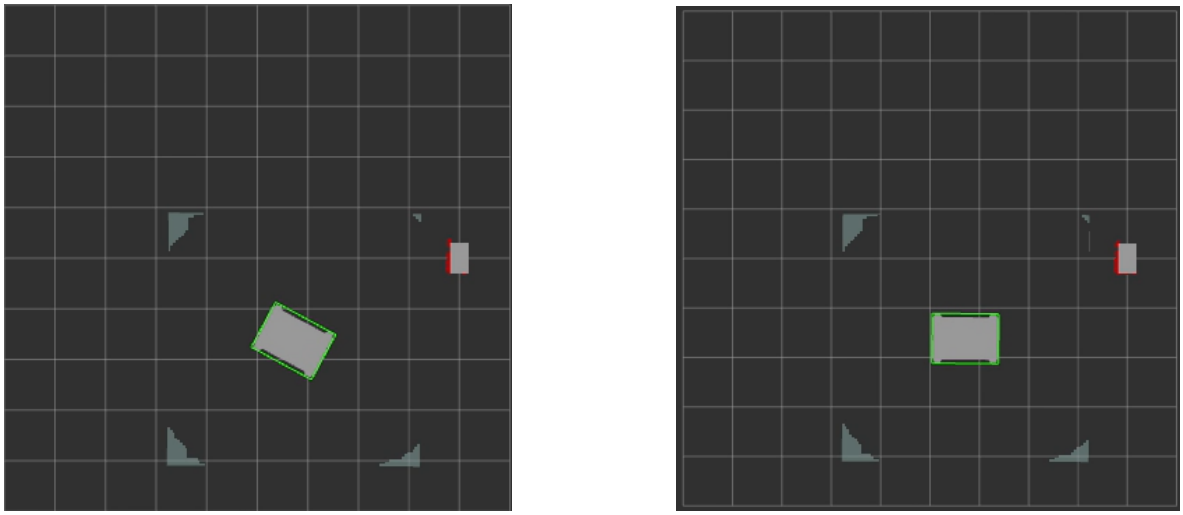


Figure 5.21: Orientation alignment process with the charger (only the yaw coordinate).

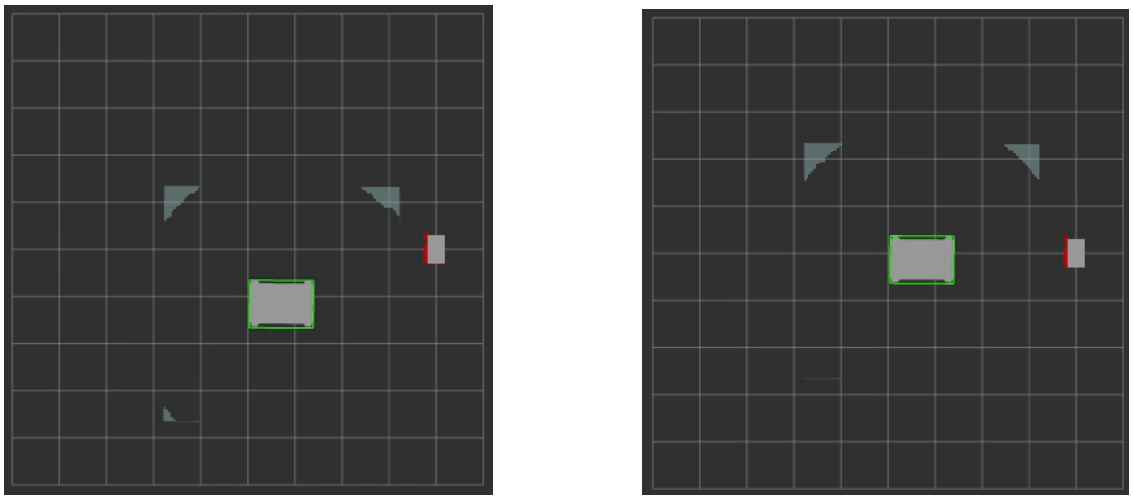


Figure 5.22: Alignment process with the charger (y-axis).

the relation between the footprint (geometric center of the robot projected on the ground) and one of the six charging pins (0.64267m), and the distance from the object (0m), defined in the configuration file - it results in an error of -0.03048 in the x-axis comparing to the defined distance from the object. The meaning of the negative connotation is that the AMR forced the pins to contract around 3mm.

5.6 Simulation Test Case: Approach to the Container

To demonstrate the flexibility of the Approach Object node, in this section, two tests are performed: approaching a container (a different object than the last section) and multiple containers. The AMR in the project CONTIGO has to approach multiple containers to execute to grasp each ordered piece and place it in the correct box located

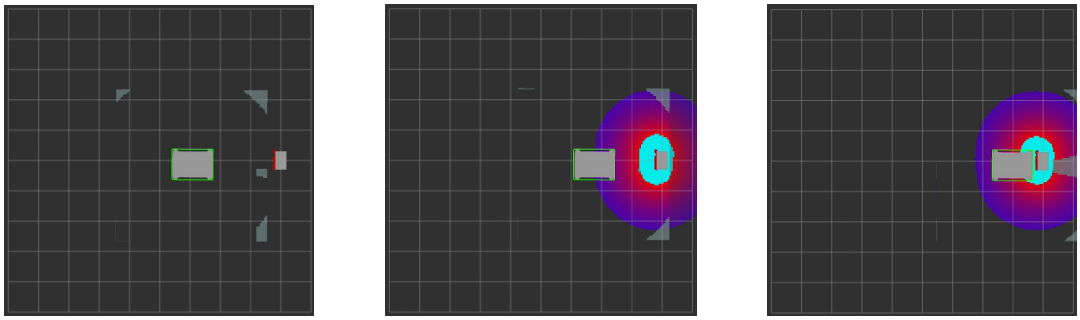


Figure 5.23: Alignment process with the charger (x-axis).

on top of the robot's structure. Thus, the next examples have a real-world application in the project.

The first test (subsection 5.6.1) aims to prove **the general application purpose of the skill *ApproachObject*** since a different object is approached (container). It also demonstrates that BTs provide a high level of actions' generalization. The second (subsection 5.6.2) has the goal of validating the flexibility of BTs and the ease of scalability that they can bring to systems.

5.6.1 Approach to a Single Container

This test case, as aforementioned, is similar to the described in section 5.5 - Approach Charging Station - because it aims to approach an object, but this time, a container.

Thus, the container's pose, ID, and type (container) are the inputs of the node. Then, the alignment algorithm is performed to position the AMR in front of the container, as shown in the following Figures 5.24, 5.25, 5.26 and 5.27.

The AMR starts, again, with a pose that differs from the container in both linear (x, y, and z) and rotational axes (row, pitch, yaw), and it should align successfully with it, performing translations and rotations.

The *main tree* code can be seen, in detail, in Listings A.6 inside appendix A, respectively. Furthermore, a demonstration video[4] is also available to better visualize the test performed.

[4] https://youtu.be/NOp6Y8BV_YE

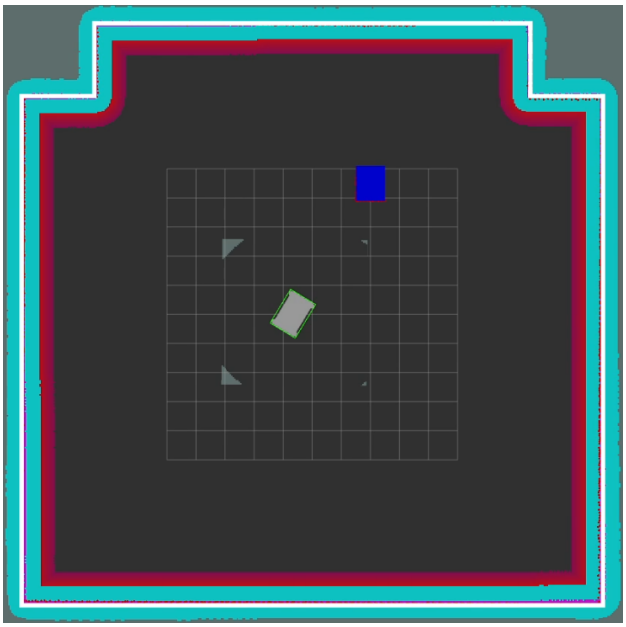


Figure 5.24: AMR's initial pose in the Approach Object Container test case. The colors near walls (white line) represent the increase in weight of trajectories as AMR get closer to them - costmap. The green (around the AMR) and blue rectangles represent the robot's footprint and the container, respectively, and the red dots around the latter are the laser beams colliding with the object.

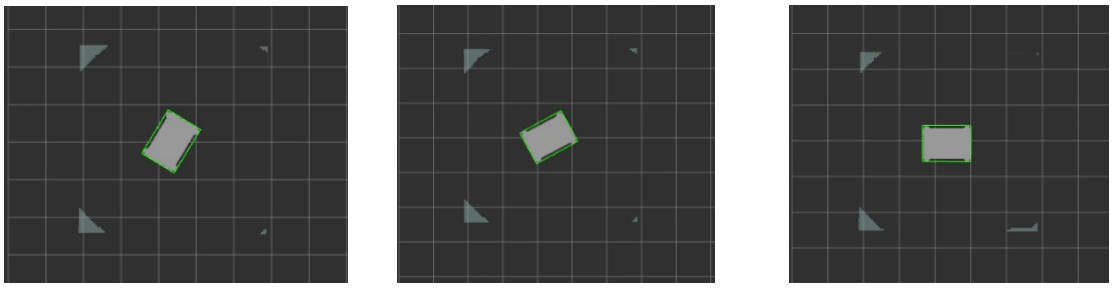


Figure 5.25: Orientation alignment process with the container (only the yaw coordinate).

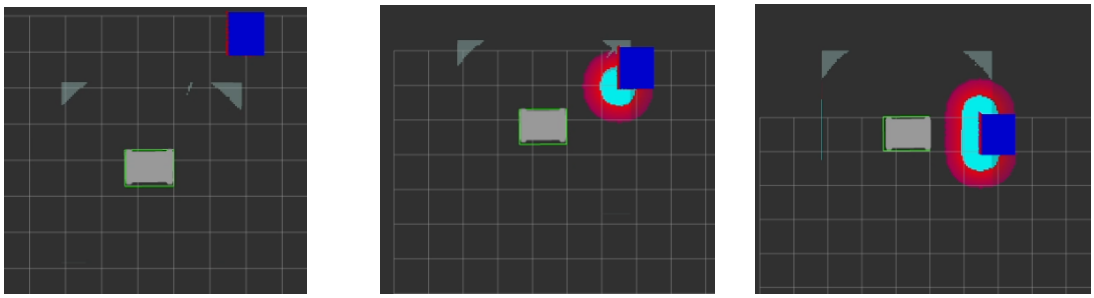


Figure 5.26: Alignment process with the container (y-axis). Ellipse with colors red and blue exhibit the inflation layer. Each color represents the weight of the area in the costmap.

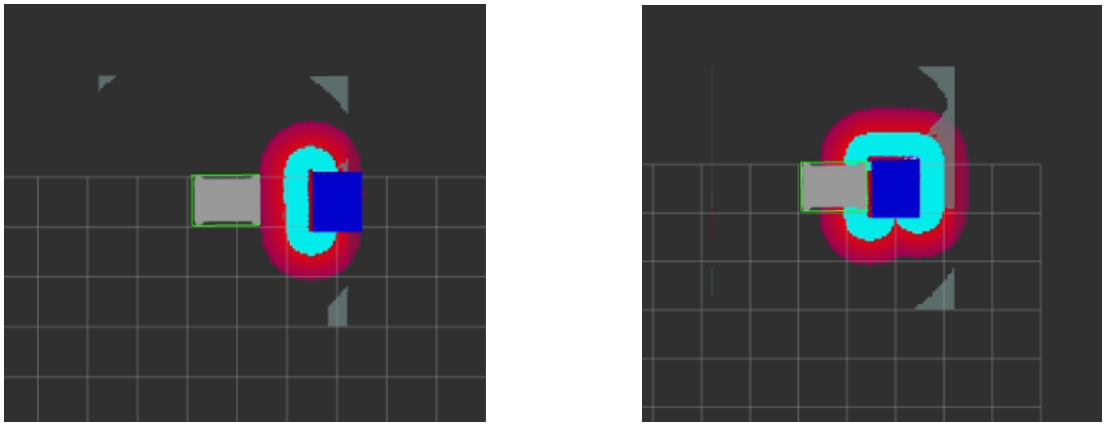


Figure 5.27: Alignment process with the container (x-axis). The distance from the object decreases progressively until the distance is equal to the one parameterized in the configuration file. The colored ellipse shows the increase in weight of the area around the object.

According to the Figure B.14, in appendix B, and the know variables - pose of the container (x=2, y=4.5, z=0, row=0, pitch=0, yaw=0), the robot's (1.3234m), and the distance from the object (0.05m), defined in the configuration file - it results and error of 0.11515m.

5.6.2 Approach to Multiple Containers

The goal of this test is to demonstrate **the scalability of Behavior Trees**. Therefore, four containers were added to the simulation. Inside appendix A are presented the Listings A.8 and A.7 to demonstrate that the difference between the two cases (approach one and multiple containers) relies on the containers' ID, pose, and the actions' organization into *SubTrees*. Moreover, only a slight change in the structure of the BT and YAML configuration file was necessary to scale the first case for multiple containers.

The main BT enclose two Sub Trees: one describe the actions needed for the robot to approach the object (identify the object's ID in the configuration file and its type and performing the approach algorithm) and the other which only have one action - to move the AMR to the home pose (the center of the map - x = 0, y = 0, z = 0 - without applied rotation - row = 0, pitch = 0 and yaw = 0).

The first *Sub Tree* named *ApproachContainerTree* starts by getting the object type from the object ID since that information is needed for the next action, which is performing the approach object algorithm. Hence, the *Approach Object* node receives as an input the object ID, type, and pose in order to execute.

Regarding the second *Sub Tree* named *MoveHomeTree*, it only consists of a single action - going to the home pose. The decision to create a subtree with only one action was with intention of not only describing better the use of the *GoToPose* node in the context of the test but also with the idea that the robot could perform more actions that could be

described in this tree, therefore it made sense to separate it.

To sum up, the behavior of the robot in the simulation is to approach each container and return to the home pose (the center of the map). Figures 5.28, B.3, 5.29 and 5.30 illustrate the sequence executed to approach the container with the ID thirty-two in the *Main Tree*, the full BT of the test, the *Sub BT* that describes the actions used to approach the object and the *Sub Tree* that describes the return to the home pose, respectively. The full tree (Figure B.3) can be better visualized in the appendix B - Figure B.3. Moreover, the Figure 5.28 was chosen to understand the sequence of approaching each container since it executed the same set of actions for all four containers except in the first one, in which the *home_pose* variable is also declared.

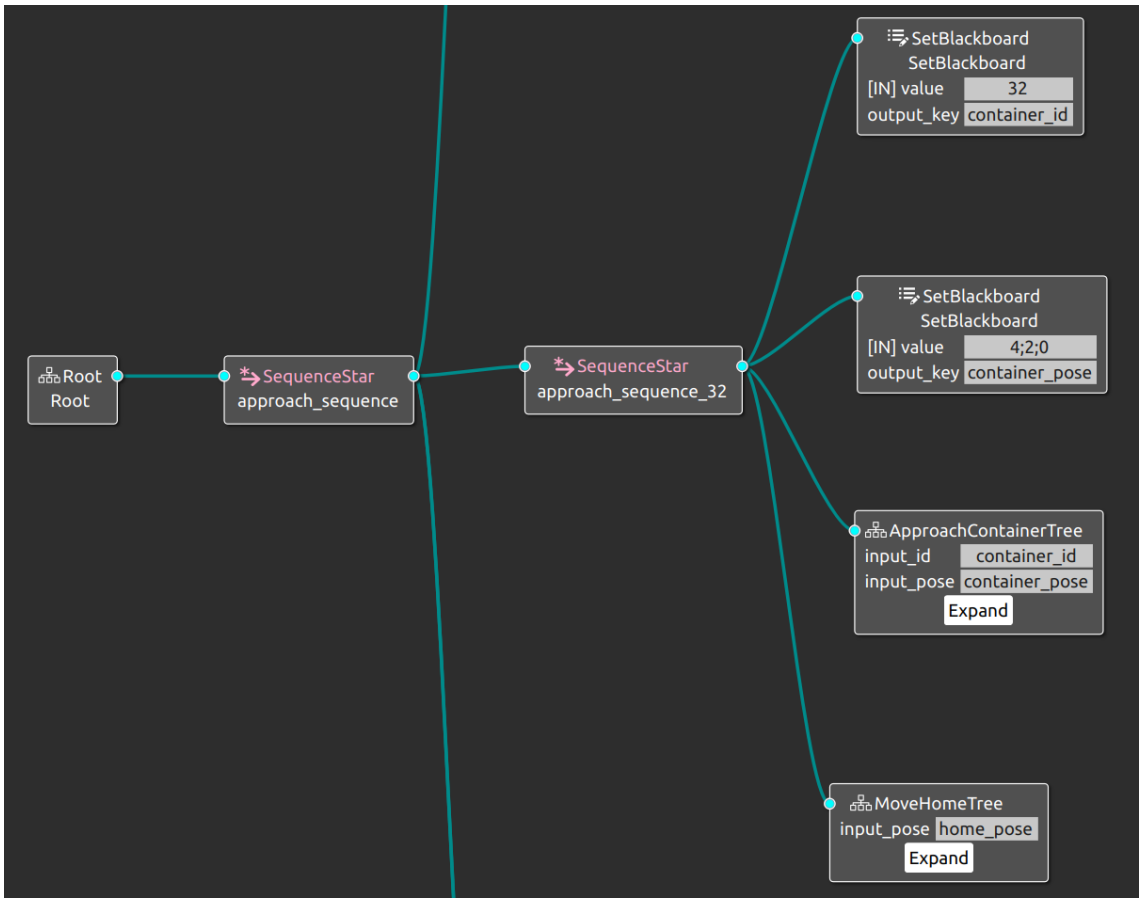


Figure 5.28: BT's branch of the container thirty-two approach sequence. Each sequence shows the type (pink-colored) and the name below. The *SetBlackboard* node is used to declare the variables *container_id* and *container_pose*. The *ApproachContainerTree* receives the *container_id* and *container_pose* variables to execute and, finally the *MoveHomeTree* is called, receiving the *home_pose* variable as input.

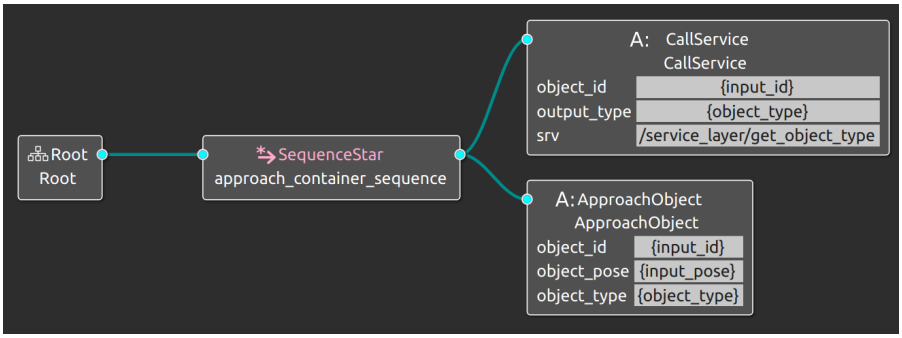


Figure 5.29: Approach Container Sub Tree. Two actions are defined in a sequence named *approach_container_sequence* with the type *sequence star*(in color pink). The *CallService* action node receives as inputs the objects ID and type and the name of the service to execute. Then the approach algorithm is performed in the node *ApproachObject* which takes in the object's ID, type, and pose as inputs.

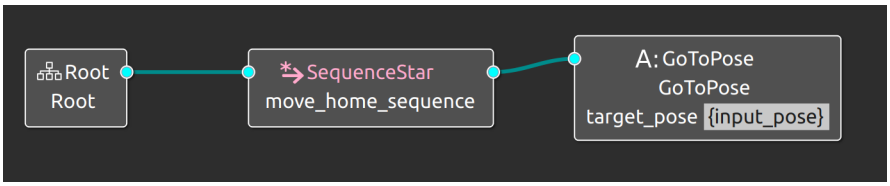


Figure 5.30: Move Home Sub Tree. This sequence named *move_home_sequence* and with the type *sequence star*, defines only one action - *GoToPose*. The action node takes the *input_pose* variable as an input.

The following figures (Figures 5.31 and 5.32) present the process of approaching one container and returning to the home pose.

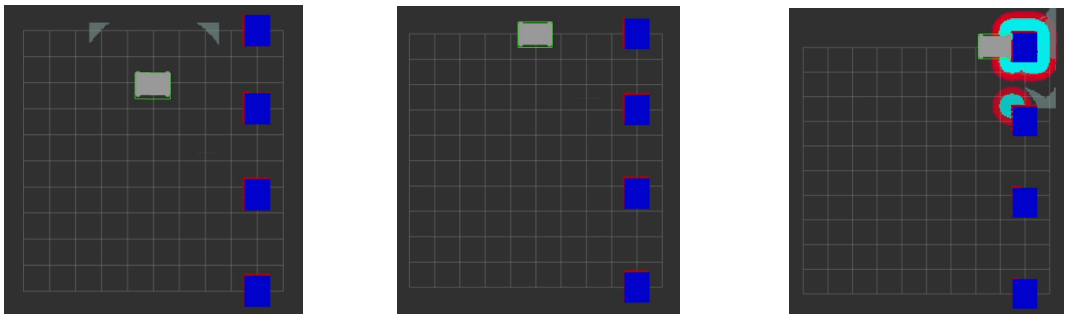


Figure 5.31: Simulation of the AMR (grey shape) approaching container number thirty-one (the first counting from upwards to downwards). The first figure (counting left to right) represents the alignment of the AMR with the container (dark blue square) on the y-axis, the second, the reduction of the distance on the x-axis, and the third, the AMR aligned with the container. The colored ellipses (light blue and red) are the inflation layer created by the costmap, the red dots around the containers are the laser scan beams, and the green square around the AMR is the robot's footprint.

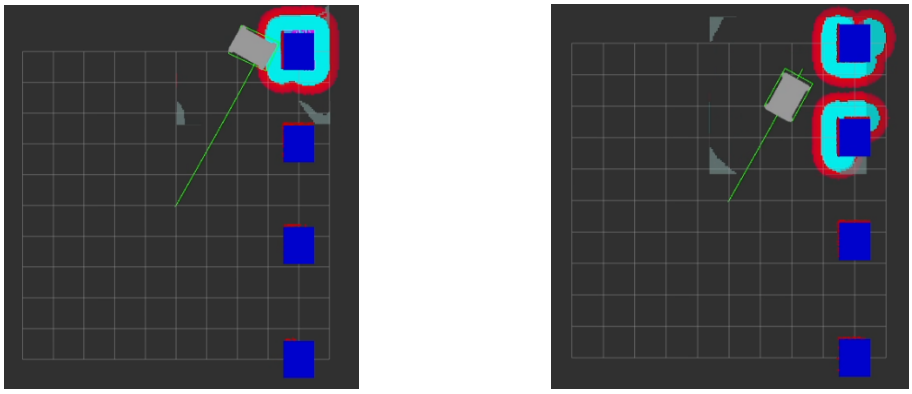


Figure 5.32: AMR returning to the *Home Position*. The green line shows the trajectory followed by the AMR (grey shape) after approaching the container with the ID number thirty-one (dark blue square). The colored ellipses (light blue and red) are the inflation layer created by the costmap, the red dots around the containers are the laser scan beams, and the green square around the AMR is the robot's footprint.

Finally, all figures regarding the test are available in the appendix B (Figures between B.4 and B.11) and a video[5] was published for demonstration purposes.

5.7 Simulation Test Case: Charge Behavior

This section presents the **validation of the *Charge* behavior**. It is composed of all the action nodes developed in this dissertation, thus, it can be considered the richest and the most complete test.

The charging procedure can be divided into the pre-approach phase and the approach phase. In the former, the first action is to localize the robot to know its position. In case it is not already in the pre-approach pose, the AMR moves in order to be prepared to approach the charger. Then, when in position, the approach algorithm is executed. The full process is depicted in the Figure 5.33.

Regarding the BT structure, the pre-approach phase is enclosed in the sub tree named *PreApproachTree* (Figure 5.34) composed by the nodes *CheckPose* and *GoToPose* and the actions of the approach phase, *CallService* and *ApproachObject*, are inside the sub tree called *ApproachChargerTree* which is similar to the tree in the Figure 5.29 only changing the value of the inputs - charger's ID and charger's pose.

Finally, the code of the *main charge BT* and its *subtrees*, in XML, is displayed in appendix A (Listings A.1 and A.2), whereas a 3D view of the AMR approaching the charger is in B (Figure B.12) and a video[6] was published to exhibit the behavior's simulation.

[5] https://youtu.be/-a8Y-C5JUw0
[6] https://youtu.be/ppbwTsfr2M0

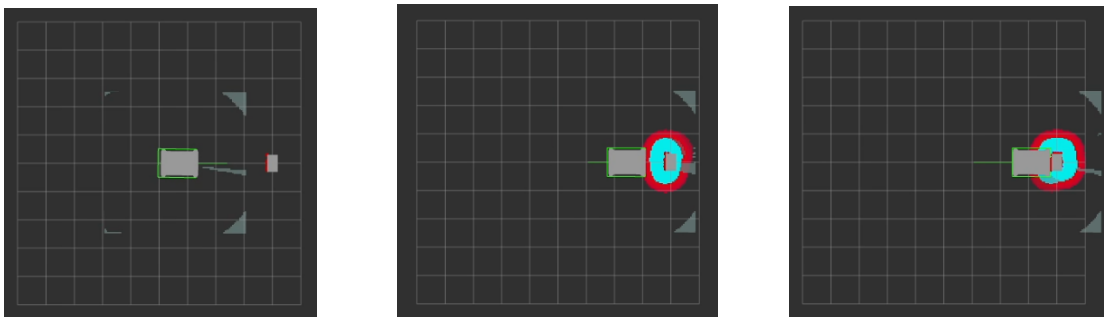


Figure 5.33: AMR (grey shape) approaching the charger (grey rectangle). In the first picture (counting left to right), the green line represents the trajectory followed by the robot to the goal pose. In the second and third pictures, the colored ellipses are the costmap's inflation layer. Since the robot navigates where the costmap has the least weight (grey space), the inflation layer creates increasing weights around obstacles, contributing to the robot's avoidance system. Lastly, the red dots around the charger are laser scan beams colliding with it, and the green rectangle around the AMR is its footprint.

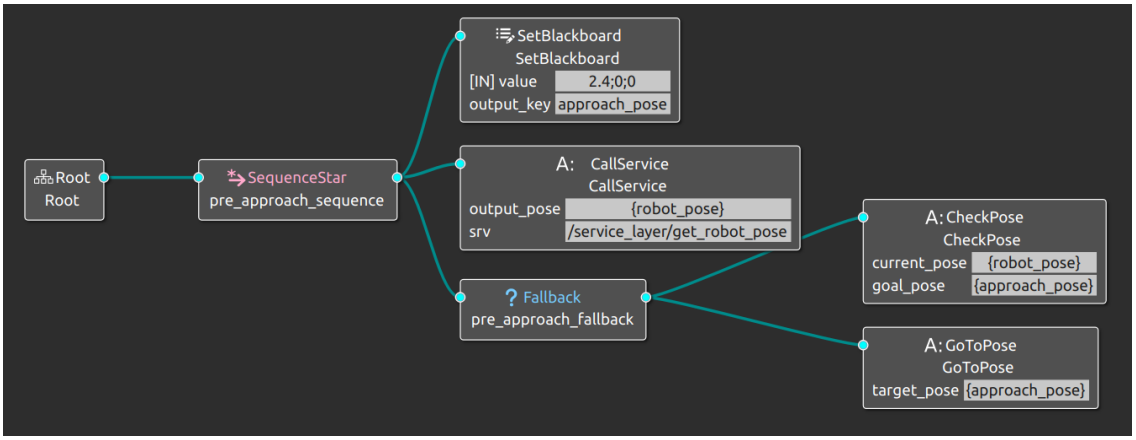


Figure 5.34: Pre Approach Sub Tree. This sequence named *pre_approach_sequence* and with the type *sequence star*(footnote), defines two actions - *SetBalckboard* and *CallService*. The former declares a variable named *approach_pose* (field output_key). The latter receives the service name as an input (*/service_layer/get_robot_pose*) and outputs the value *robot_pose*. The other branch is a fallback sequence where two actions are executed the *CheckPose* which receives two inputs - current_pose and goal_pose - and *GoToPose* that receives only one - target_pose.

5.8 BT Implementation vs. ROS Node Implementation

This section aims to **compare the performance of the BT's implementation against a ROS node implementation timewise**. In other words, the former, regarding the test case *approach multiple containers* (Section 5.6.2), was centralized in a single ROS node to compare the time it takes to complete the full process (approach four containers and return to the home pose). The *approach multiple containers* test case was chosen for this comparison as it is the longest process, thus, the real impact of communication times

between the implemented architecture and ROS and ROS nodes' interaction itself can be uncovered.

It is important to mention that the laptop used is the one described in detail in Section 5.2. There weren't any applications running in the background while the simulation was running (other than the internal processes of the operating system), and the computer was restarted between each test to guarantee that most amount of the computer's resources were available for the simulation.

Five full processes were run for each implementation, and the results were disclosed in the following table.

Table 5.1: Time spent (in minutes) on each kind of implementation in the full process to approach four containers.

Attempts	**ROS Node**	**Behavior Tree Nodes**
1st	02:29	02:40
2nd	02:26	02:38
3rd	02:24	02:29
4th	02:27	02:32
5th	02:27	02:33
Avg	02:27	02:34

From the table, it can be concluded that, on average, the *BT Node* implementation takes more 7 seconds than the *ROS Node*.

The results of each test are analyzed and discussed in the next section (Section 5.9).

5.9 Experimental Results' Discussion

This section aims to analyze and discuss each of the tests presented in the previous sections.

Firstly, **the first two cases** (Sections 5.3 and 5.4) were simple and used to validate the overall functioning of the system. Hence, it can be perceived that the implemented layered architecture (Task Layer, Skill Layer, and Interface Layer) can operate with an external system capable of providing the necessary feedback, in this case, ROS. One of the most important skills developed, the *GoToPose*, which provides the communication with ROS navigation stack, proved to work fluently and successfully in all the simulations.

It can also be understood the **flexibility** and **independency** of the skills with the figures showing the BT's structure of the tests (Figures 5.9 and 5.13), since some nodes have several inputs and outputs which are used according to the situation without being mandatory. For example, the skill *CallService*, when the service name is */get_robot_pose*, the output is a pose whereas in case of the name being */get_object_type*, it outputs an integer (referring to an object type) - flexibility.

Regarding independence, all the skills are independent of each other, meaning that they can be integrated and work in another system as long as the necessary inputs are provided. That's the goal of BT actions. However, in this dissertation, some work had to be done outside of the proposed architecture, meaning creating a custom action server node to perform the approach algorithm. ROS possess some inner mechanisms that have to be carefully approached when designing fully independent solutions. Custom ROS service and action's servers (entities that doesn't exist by default in ROS) have to run inside ROS nodes. For instance, there is the ROS navigation stack developed for mobile robots by default, which has action servers already ready to receive requests from external nodes. In that case, it's not necessary to develop a custom server to interface with the robot's mobile base.

Therefore, there is no guarantee that, when developing a BT node, it will work in another robotic system running ROS if the appropriate servers are not running. For example, the *ApproachObject* skill make a request to an action server called *approach_node*. Consequently, it was created a ROS node (*ApproachNode*) running the *approach_node* action server, otherwise it would not be possible to link the BT action with ROS. Despite that, the majority of the mobile robots running ROS use the navigation stack for navigation, its inner mechanisms and packages developed to work with it, hence the advantages provided by this architecture outweigh the possible few adjustments to be made.

In relation to **the third** (Section 5.5) **and fourth** (Section 5.6.1) **tests**, it is recognized another advantage Behavior Trees - **modularity**. With the same node (*ApproachObject*) and a slight modification (Figures A.5 and A.6), it was proved that different types of objects defined in the configuration file of the system (YAML file) could be approached. Particularly, two object types were approached successfully (a charger and a container) within the expected error margins (Figures B.13 and B.14) in appendix B). Both errors resulting from the tests would not have had a great impact in the real world as the charger pins are flexible (meaning that they can contract about 2cm), and the software of the robotic arm, in the AMR, can adjust its trajectories of picking the pieces as long as it's not too far away from the container, in the x and y-axis.

Respecting **the fifth example** (Section 5.6.2), it can be understood the easiness of **scale** and **modify** BTs. Comparing the configuration files from the simulation case where a single container is approached (Listing A.9) and the approach multiple containers examples (A.10), the only step necessary was to define 3 more containers with different, define their dimensions and parameters for the approach algorithm to use. The same method would be valid for adding more charging stations or other objects that the developer wants to define. On the side of the Behavior Tree, the only required change is to the ID number of the object. Moreover, in the Figure B.3, the complexity does not compromise the readability of the tree because it is still clear the flow of the process to be executed. It is clear that the mentioned BT could be more simplified. For instance, create a loop where in each sequence, the input would be the container's ID and pose. Then the four branches would turn into one.

Similarly, in the **Charge Behavior Case** (Section 5.7), different actions were encapsulated into two *SubTrees*, resulting in a behavior easier to read and understand. Thus, BTs can enhance the process of designing behaviors, making them easier to grow and from the outside (more generic) to the inside (more specific).

Finally, **comparing two different implementations** (Section 5.8), it can be perceived one of the most discussed disadvantage of Behavior Trees - **performance**. From the Figure 5.1 can be observed that the BT implementation took longer to execute the full process, in every attempt, with the average time being 7 seconds more than the full process implemented in only one ROS node. It was also noticed that the BT implementation took longer to initialize before the system started to communicate and the simulation to run.

Consequently, some considerations can be concluded. The extra time taken by the BT implementation can be due to the data's exchange between layers, which is not a concern when it is considered a centralized approach (ROS node) since the code runs sequentially with shorter waiting times between processes. Hence, due to longer communication times, occurrences of timeouts waiting for the transforms, necessary to the *ROS Navigation Stack*, were triggered, in four of the five attempts regarding the BT node implementation, resulting in simulation delay.

ROS is, intrinsically, a multi-threaded system with increasingly data throughput, which makes, on a larger scale, resource demanding. On the other hand, Behavior Tree are also computationally demanding due to their inner processes and default abstraction level, making it easier to load numerous nodes with the tendency, in more complex systems, to overload them. Therefore, combining ROS and Behavior Trees can be challenging performance wise. However, comparing the advantages with the disadvantages, it can be perceived that the majority of the robotic systems can improve, being more modular, more flexible, and easily and rapidly modified.

Returning to the comparison of the duration for each test, seven seconds has little to no impact when it comes to robot navigation. It can be easily mitigated increasing the velocity of the robot or introducing a more powerful hardware. However, it can be a become a problem in computationally modest to weak systems. For instance, regarding critical systems (which the majority does not run ROS), robots in factories and big companies exist to automate and increase the performance of the production. If a robot takes more than a human to complete a process or compromises the assembly line, it can lead to money loss or even accidents since an assembly line is composed of a variety of robots. Thus, in some systems, it is mandatory to extensively test the BT implementation in order to adopt it or even abandon this option because of the resource demand of such application.

To conclude, BTs possess some advantages for implementing robot behaviors. They provide modularity, which is beneficial for reusing code and behavior tree design across different systems and adding, removing, or transforming modules easily, and some level of independence that needs to be approached and analyzed between system to system, that is, when working with ROS BTs is not a fully independent entity which can be added

or removed to the system without any consideration. The capacity of encapsulating basic actions into sub-processes (*SubTrees*), visualize, and increase complexity without a compromising readability is an improvement over other methodologies. However, these advantages come with a price - performance. BTs are computationally demanding and can cause problems in critical systems, thus, it is important to measure applicability of them in each system.

6

Conclusions and Future Work

This chapter summarises the goals achieved in this dissertation, discusses the proposed approach, and compares the intentions of the implementation and comparing the results obtained. Furthermore, some suggestions to take into consideration for future research are outlined.

6.1 Conclusions

A behavior-based autonomous mobile robot was presented. The design of Behavior Trees to define a set of actions, and the advantages of such structures began to spread through the gaming industry. Shortly, other industries and research fields, like robotics, began to recognize them as well. The implementation of the researched architecture proposed by Colledanchise and Natale [76] was developed with slight modifications to accommodate the link between ROS, the main operating system in robotics research, to accelerate the development of full robotic solutions and such architecture.

The proposed architecture consisted of three layers hierarchically structured by the level of abstraction: the *Task Layer* (the Behavior Trees itself), the *Skill Layer* (set of generic action nodes), and the *Interface Layer* (communication interface between the based layered approach and ROS and named by the aforementioned researchers as *Service Layer*). Both the *Task layer* and *Skill Layer* were developed accordingly to the approach followed by Colledanchise and Natale [76], however, the *Interface Layer* was enhanced to meet some ROS inner logistics. Despite affirming the impossibility of developing a centralized *Interface Layer*, in this dissertation, such layer was composed of two nodes: the *Service Node* and the *Approach Node*. The former was used to communicate with ROS services and default action servers, and the latter handled the approach node *action server*. Therefore, it was concluded that ROS custom action servers need their own nodes to run.

Four action nodes were developed to show the capabilities of BTs in a robotic system. More behaviors were planned to be implemented, for example, the pick and place, by developing the grasping BT action and creating nodes to make unitary trajectories for robotic arms, which would be a great modular solution for robot manipulators. Despite

this, such nodes proved their usability in a simulated environment, from the simplest to more complex actions. All the simulated scenarios were completed successfully, the ability to create a behavior for charging the AMR was validated as well as his ability to approach known objects present in the loaded configuration file. Hence, the communication with ROS was successfully achieved with the developed architecture, and was possible to analyze some real results.

The seven test cases evidenced the advantages of the BT approach to create behaviors and the possible drawbacks. Firstly, it was noticed the flexibility provided by such approach with the number of test scenarios created, combining the developed actions in different sequences. It was also perceived the increasing modularity and independence of BT nodes when considering complex systems. Particularly, the former characteristic was revealed by using exactly the same node (*Approach Node*) to approach two different objects. Hence, the only change required to switch from approaching a charging station to approaching a container was just an adjustment of the required inputs. Regarding independence, it was perceived that skills (action nodes) achieve some level of independence but do not necessarily need each other to work properly. Nonetheless, skills that require communication with ROS action servers must be carefully analyzed before being considered fully independent of the system.

Behavior Trees also exhibited that they can accelerate the process of scalability in some complex systems. For instance, from approaching one container to approaching four, it was only necessary to add more three containers in the YAML file for it to perform the action for all of them. This characteristic can be a real advantage when considering ever-changing environments where new entities are being added to the environment and the robots need to consider those changes rapidly. Furthermore, developing new solutions from the previously existing ones can improve drastically timewise. This idea of agile scalability also can require less savvy individuals to proceed with the required changes. Lastly, the final test case concluded that all these advantages come with the cost - performance. Behavior Trees are computationally heavy to process, thus, the delay can increase the longer the process is. Communication delays between different layers were also verified compared to a simple implementation via only one ROS node due to the intrinsic extra load of processing the trees. Although these symptoms can be mitigated with more resourceful systems, integrating a BT solution in a modest to low-end system has to be carefully considered.

In conclusion, the proposed approach offered useful insights into the advantages, drawbacks, use cases, and conditions to implement such systems. Further real-world tests, with AMR of the project CONTIGO, [11] need to be executed, but simulations displayed beneficial results for scalability and rapid implementation of new robotic behaviors when considering changing environments. In terms of applicability, the AMR is going to perform its tasks in the Volkswagen Autoeuropa shopfloor where BTs can have a real benefit, although delays can have an impact security-wise, hence, those aspects have thought about.

6.2 Future Work

Following the guidelines of Behavior Trees, the work developed can be deeply explored, but it also opened multiple opportunities for new researchers, based on the results, namely:

- Validate other skills and behaviors in terms of usability and scalability. In the industry, these two characteristics are of high importance to transition from a prototype to a product, hence, taking a research topic into real-world use cases;
- Automatically create Behavior Trees from natural language descriptions;
- Validate the full solution of pick and place in a real-world industrial environment and study the impact of possible overloads in the systems with the proposed architecture.
- Test the resilience of the architecture by creating multiple simulation worlds exploiting domain randomization;
- Adapt Behavior Trees to perform human-aware navigation.

Bibliography

[1] A. G. Frank, L. S. Dalenogare, and N. F. Ayala. "Industry 4.0 technologies: Implementation patterns in manufacturing companies". In: *International Journal of Production Economics* 210 (2019), pp. 15–26. DOI: 10.1016/j.ijpe.2019.01.004 (cit. on p. 1).

[2] A. Nayyar and A. Kumar. *A roadmap to industry 4.0: Smart production, sharp business and sustainable development*. Springer, 2020, p. 7. ISBN: 978-3-030-14546-0. DOI: 10.1007/978-3-030-14544-6 (cit. on p. 1).

[3] *Association for Advancing Automation*. URL: https://www.automate.org. (accessed: 24.09.2022) (cit. on p. 1).

[4] *Q3 Robot Orders Put 2021 on Track for Biggest Year Yet*. URL: https://www.automate.org/news/q3-robot-orders-put-2021-on-track-for-biggest-year-yet. (accessed: 24.09.2022) (cit. on p. 1).

[5] *Robot sales surge in Europe, Asia and the Americas*. URL: https://ifr.org/ifr-press-releases/news/robot-sales-surge-in-europe-asia-and-the-americas. (accessed: 24.09.2022) (cit. on p. 1).

[6] *AGV (Automated Guided Vehicles) and AMR (Autonomous Mobile Robots) Market Opportunity worth more than $18B by 2027 with an installed base of 2.4 Million Robots - Driven by Logistics Manufacturing, Market Forecast till 2027*. URL: https://www.researchandmarkets.com/reports/5398204/agv-automated-guided-vehices-and-amr?utm_source=GNOM&utm_medium=PressRelease&utm_code=hb8lpt&utm_campaign=16636+-+Global+AGV+(Automated+Guided+Vehicles)+and+AMR+(Autonomous+Mobile+Robots)+Market+Opportunity+worth+more+than+%5C%2418B+by+2027&utm_exec=elco286prd. (accessed: 24.09.2022) (cit. on p. 1).

[7] H. Mitta, D. Chandra, and S. Kumar. "Reducing power in using different technologies using FSM architecture". In: *International Journal of VLSI Design & Communication Systems* 2.3 (2011), p. 243. DOI: 10.5121/vlsic.2011.2320 (cit. on p. 2).

[8] A. R. Cavalli, T. Higashino, and M. Núñez. "A survey on formal active and passive testing with applications to the cloud". In: *annals of telecommunications-annales des télécommunications* 70.3 (2015), pp. 85–93. DOI: 10.1007/s12243-015-0457-8 (cit. on p. 2).

[9] M. Di Angelo and G. Salzer. "A survey of tools for analyzing Ethereum smart contracts". In: *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. IEEE. 2019, pp. 69–78. DOI: 10.1109/DAPPCON.2019.00018 (cit. on p. 2).

[10] M. Colledanchise and P. Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018 (cit. on pp. 2, 16, 20, 22).

[11] *CONTIGO*. URL: https://contigo.introsys.eu/. (accessed: 26.08.2022) (cit. on pp. 2, 4, 47, 80).

[12] *INTROSYS - Global Control System Designers*. URL: https://introsys.eu/. (accessed: 26.08.2022) (cit. on pp. 2, 4, 47).

[13] *Volkswagen Autoeuropa*. URL: https://www.volkswagenautoeuropa.pt/. (accessed: 26.08.2022) (cit. on pp. 2, 47).

[14] F. Rovida, B. Grossmann, and V. Krüger. "Extended behavior trees for quick definition of flexible robotic tasks". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 6793–6800. DOI: 10.1109/IROS.2017.8206598 (cit. on p. 2).

[15] B. Banerjee. "Autonomous acquisition of behavior trees for robot control". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 3460–3467. DOI: 10.1109/IROS.2018.8594083 (cit. on p. 3).

[16] J. Styrud et al. "Combining Planning and Learning of Behavior Trees for Robotic Assembly". In: *arXiv preprint arXiv:2103.09036* (2021) (cit. on p. 3).

[17] M. Colledanchise and L. Natale. "Analysis and exploitation of synchronized parallel executions in behavior trees". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2019, pp. 6399–6406. DOI: 10.1109/IROS40897.2019.8967812 (cit. on p. 3).

[18] E. Giunchiglia et al. "Conditional behavior trees: Definition, executability, and applications". In: *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*. IEEE. 2019, pp. 1899–1906. DOI: 10.1109/SMC.2019.8914358 (cit. on p. 3).

[19] H. Hu et al. "Self-adaptive traffic control model with Behavior Trees and Reinforcement Learning for AGV in Industry 4.0". In: *IEEE Transactions on Industrial Informatics* 17.12 (2021), pp. 7968–7979. DOI: 10.1109/TII.2021.3059676 (cit. on p. 3).

[20] F. Fusaro et al. "A human-aware method to plan complex cooperative and autonomous tasks using behavior trees". In: *2020 IEEE-RAS 20th International Conference on Humanoid Robots (Humanoids)*. IEEE. 2021, pp. 522–529. DOI: 10.1109/HUMANOIDS47582.2021.9555683 (cit. on p. 3).

[21] *RobMoSys Wiki*. URL: https://robmosys.eu/wiki/start. (accessed: 24.09.2022) (cit. on p. 3).

[22] URL: https://www.volkswagenautoeuropa.pt/. (accessed: 23.01.2022) (cit. on p. 4).

[23] *BehaviorTree.CPP*. URL: https://www.behaviortree.dev/. (accessed: 08.01.2022) (cit. on p. 4).

[24] A. Filotheou et al. "Quantitative and qualitative evaluation of ROS-enabled local and global planners in 2D static environments". In: *Journal of Intelligent & Robotic Systems* 98.3 (2020), pp. 567–601. DOI: 10.1007/s10846-019-01086-y (cit. on p. 8).

[25] P. Marin-Plaza et al. "Global and local path planning study in a ROS-based research platform for autonomous vehicles". In: *Journal of Advanced Transportation* 2018 (2018). DOI: 10.1155/2018/6392697 (cit. on p. 8).

[26] K. Cai et al. "Mobile robot path planning in dynamic environments: a survey". In: *arXiv preprint arXiv:2006.14195* (2020) (cit. on p. 8).

[27] E. W. Dijkstra et al. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271 (cit. on pp. 8, 9).

[28] F. Duchoň et al. "Path planning with modified a star algorithm for a mobile robot". In: *Procedia Engineering* 96 (2014), pp. 59–69. DOI: 10.1016/j.proeng.2014.12.098 (cit. on p. 8).

[29] C. Wang and M. Q.-H. Meng. "Variant step size RRT: An efficient path planner for UAV in complex environments". In: *2016 IEEE International Conference on Real-time Computing and Robotics (RCAR)*. IEEE. 2016, pp. 555–560. DOI: 10.1109/RCAR.2016.7784090 (cit. on p. 8).

[30] W. Chi et al. "Risk-DTRRT-based optimal motion planning algorithm for mobile robots". In: *IEEE Transactions on Automation Science and Engineering* 16.3 (2018), pp. 1271–1288. DOI: 10.1109/TASE.2018.2877963 (cit. on p. 8).

[31] X. Dai et al. "Mobile robot path planning based on ant colony algorithm with A* heuristic method". In: *Frontiers in neurorobotics* 13 (2019), p. 15. DOI: 10.3389/fnbot.2019.00015 (cit. on p. 8).

[32] T. T. Mac et al. "A hierarchical global path planning approach for mobile robots based on multi-objective particle swarm optimization". In: *Applied Soft Computing* 59 (2017), pp. 68–76. DOI: 10.1016/j.asoc.2017.05.012 (cit. on pp. 8, 9).

[33] C. Lamini, S. Benhlima, and A. Elbekri. "Genetic algorithm based approach for autonomous mobile robot path planning". In: *Procedia Computer Science* 127 (2018), pp. 180–189. DOI: 10.1016/j.procs.2018.01.113 (cit. on p. 8).

[34] X. Wang et al. "Double global optimum genetic algorithm–particle swarm optimization-based welding robot path planning". In: *Engineering Optimization* 48.2 (2016), pp. 299–316. DOI: 10.1080/0305215X.2015.1005084 (cit. on p. 9).

[35] S. W. AbuSalim et al. "Comparative analysis between dijkstra and bellman-ford algorithms in shortest path optimization". In: *IOP Conference Series: Materials Science and Engineering*. Vol. 917. 1. IOP Publishing. 2020, p. 012077. DOI: 10.1088/1757-899x/917/1/012077 (cit. on p. 10).

[36] *global$_p$lanner*. URL: http://wiki.ros.org/global_planner. (accessed: 11.02.2022) (cit. on pp. 11, 12).

[37] *navfn*. URL: http://wiki.ros.org/navfn. (accessed: 11.02.2022) (cit. on pp. 11, 56).

[38] O. Brock and O. Khatib. "High-speed navigation using the global dynamic window approach". In: *Proceedings 1999 ieee international conference on robotics and automation (Cat. No. 99CH36288C)*. Vol. 1. IEEE. 1999, pp. 341–346. DOI: 10.1109/ROBOT.1999.770002 (cit. on p. 11).

[39] R. Philippsen. *Motion planning and obstacle avoidance for mobile robots in highly cluttered dynamic environments*. Tech. rep. EPFL, 2004. DOI: 10.5075/epfl-thesis-3146 (cit. on p. 12).

[40] D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance". In: *IEEE Robotics & Automation Magazine* 4.1 (1997), pp. 23–33 (cit. on p. 12).

[41] C. Rösmann. "Difference between DWA and TEB local planners". In: (2018) (cit. on p. 12).

[42] *dwa$_l$ocal$_p$lanner*. URL: http://wiki.ros.org/dwa_local_planner. (accessed: 17.02.2022) (cit. on p. 13).

[43] S. Quinlan and O. Khatib. "Elastic bands: Connecting path planning and control". In: *[1993] Proceedings IEEE International Conference on Robotics and Automation*. IEEE. 1993, pp. 802–807 (cit. on p. 13).

[44] F. Pimentel and P. Aquino. "Performance evaluation of ROS local trajectory planning algorithms to social navigation". In: *2019 Latin American Robotics Symposium (LARS), 2019 Brazilian Symposium on Robotics (SBR) and 2019 Workshop on Robotics in Education (WRE)*. IEEE. 2019, pp. 156–161. DOI: 10.1109/LARS-SBR-WRE48964.2019.00035 (cit. on pp. 14, 15).

[45] F. d. A. M. Pimentel and P. T. Aquino-Jr. "Evaluation of ROS Navigation Stack for Social Navigation in Simulated Environments". In: *Journal of Intelligent & Robotic Systems* 102.4 (2021), pp. 1–18. DOI: 10.1007/s10846-021-01424-z (cit. on pp. 14, 15).

[46] Ö. Ararat and B. A. Güvenç. "Development of a collision avoidance algorithm using elastic band theory". In: *IFAC Proceedings Volumes* 41.2 (2008), pp. 8520–8525. DOI: 10.3182/20080706-5-KR-1001.01440 (cit. on p. 16).

[47] C. Rösmann, F. Hoffmann, and T. Bertram. "Planning of multiple robot trajectories in distinctive topologies". In: *2015 European Conference on Mobile Robots (ECMR)*. IEEE. 2015, pp. 1–6. DOI: 10.1109/ECMR.2015.7324179 (cit. on p. 16).

[48] *Simple state machine on Unity*. URL: https://sudonull.com/post/117825-Simple-state-machine-on-Unity. (accessed: 17.01.2022) (cit. on p. 17).

[49] M. Iovino et al. "A survey of behavior trees in robotics and ai". In: *arXiv preprint arXiv:2005.05842* (2020) (cit. on pp. 20, 21).

[50] E. Adam and R. Mandiau. "Roles and hierarchy in multi-agent organizations". In: *International Central and Eastern European Conference on Multi-Agent Systems*. Springer. 2005, pp. 539–542. DOI: 10.1007/11559221_55 (cit. on p. 21).

[51] S. Macenski et al. "The Marathon 2: A Navigation System". In: *arXiv preprint arXiv:2003.00368* (2020) (cit. on pp. 21, 22).

[52] J. A. Bagnell et al. "An integrated system for autonomous robotics manipulation". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 2955–2962. DOI: 10.1109/IROS.2012.6385888 (cit. on p. 22).

[53] K. R. Guerin et al. "A framework for end-user instruction of a robot assistant for manufacturing". In: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2015, pp. 6167–6174. DOI: 10.1109/ICRA.2015.7140065 (cit. on pp. 22, 24).

[54] A. Klöckner. "Behavior trees for UAV mission management". In: *INFORMATIK 2013: informatik angepasst an Mensch, Organisation und Umwelt* (2013), pp. 57–68 (cit. on p. 22).

[55] A. Klöckner. "The Modelica BehaviorTrees Library: Mission planning in continuous-time for unmanned aircraft". In: *Proceedings of the 10th International Modelica Conference-Lund, Sweden-Mar 10-12, 2014*. 96. Linköping University Electronic Press. 2014, pp. 727–736. ISBN: 978-91-7519-380-9 (cit. on p. 22).

[56] R. H. Abiyev, N. Akkaya, and E. Aytac. "Control of soccer robots using behaviour trees". In: *2013 9th Asian Control Conference (ASCC)*. IEEE. 2013, pp. 1–6. DOI: 10.1109/ASCC.2013.6606326 (cit. on p. 22).

[57] R. Ghzouli et al. "Behavior trees in action: a study of robotics applications". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. 2020, pp. 196–209. DOI: 10.1145/3426425.3426942 (cit. on pp. 23–25).

[58] *BehaviorTree.CPP library Documentation*. URL: https://www.behaviortree.dev/. (accessed: 04.02.2022) (cit. on pp. 23, 24).

[59] *PyTrees library Documentation*. URL: https://py-trees.readthedocs.io/en/devel/. (accessed: 04.02.2022) (cit. on pp. 23, 24).

[60] *PyTrees ROS library Documentation*. URL: https://py-trees-ros.readthedocs.io/en/devel/. (accessed: 04.02.2022) (cit. on pp. 23, 24).

[61] *CoSTAR library Documentation*. URL: http://cpaxton.github.io/costar_stack/. (accessed: 04.02.2022) (cit. on pp. 23, 24).

[62] *Unreal Engine library Documentation*. URL: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/. (accessed: 04.02.2022) (cit. on p. 23).

[63] *ROS*. URL: https://www.ros.org/. (accessed: 27.09.2022) (cit. on p. 27).

[64] T. Elmokadem and A. V. Savkin. "A method for autonomous collision-free navigation of a quadrotor UAV in unknown tunnel-like environments". In: *Robotica* 40.4 (2022), pp. 835–861. DOI: 10.1017/S0263574721000849 (cit. on p. 27).

[65] D. Guffanti, A. Brunete, and M. Hernando. "Development and validation of a ROS-based mobile robotic platform for human gait analysis applications". In: *Robotics and Autonomous Systems* 145 (2021), p. 103869. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2021.103869 (cit. on p. 27).

[66] R. Rahimi et al. "An Industrial Robotics Application with Cloud Computing and High-Speed Networking". In: *2017 First IEEE International Conference on Robotic Computing (IRC)*. 2017, pp. 44–51. DOI: 10.1109/IRC.2017.39 (cit. on p. 27).

[67] S. Lagraa et al. "Real-Time Attack Detection on Robot Cameras: A Self-Driving Car Application". In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. 2019, pp. 102–109. DOI: 10.1109/IRC.2019.00023 (cit. on p. 27).

[68] A. Giaretta, M. De Donno, and N. Dragoni. "Adding Salt to Pepper: A Structured Security Assessment over a Humanoid Robot". In: *Proceedings of the 13th International Conference on Availability, Reliability and Security*. Association for Computing Machinery, 2018. ISBN: 9781450364485. DOI: 10.1145/3230833.3232807 (cit. on p. 27).

[69] *ROS.org: Documentation*. URL: http://wiki.ros.org/Documentation. (accessed: 27.09.2022) (cit. on p. 27).

[70] *What is ROS?* URL: https://roboticsbackend.com/what-is-ros/. (accessed: 20.05.2022) (cit. on p. 27).

[71] *ROS 2 Documentation*. URL: https://docs.ros.org/en/humble/. (accessed: 30.09.2022) (cit. on p. 27).

[72] S. Rivera et al. "ROS-Defender: SDN-Based Security Policy Enforcement for Robotic Applications". In: *2019 IEEE Security and Privacy Workshops (SPW)*. 2019, pp. 114–119. DOI: 10.1109/SPW.2019.00030 (cit. on p. 28).

[73] *move_base*. URL: http://wiki.ros.org/move_base. (accessed: 30.07.2022) (cit. on p. 28).

[74] *Understanding services*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html. (accessed: 05.08.2022) (cit. on p. 31).

[75] *Understanding actions*. URL: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html. (accessed: 05.08.2022) (cit. on p. 32).

[76] M. Colledanchise and L. Natale. "On the Implementation of Behavior Trees in Robotics". In: *IEEE Robotics and Automation Letters* (2021). DOI: 10.1109/LRA.2021.3087442 (cit. on pp. 33, 79).

[77] *behavior_stack_example*. URL: https://github.com/hsp-iit/behavior-stack-example. (accessed: 15.08.2022) (cit. on p. 36).

[78] *Groot*. URL: https://github.com/BehaviorTree/Groot. (accessed: 21.08.2022) (cit. on pp. 51, 56).

[79] *GAZEBO*. URL: https://gazebosim.org/home. (accessed: 23.09.2022) (cit. on p. 55).

[80] *rviz*. URL: http://wiki.ros.org/rviz. (accessed: 23.09.2022) (cit. on p. 55).

[81] *teb_local_planner*. URL: http://wiki.ros.org/teb_local_planner. (accessed: 23.09.2022) (cit. on p. 56).

A

Appendix A - Behavior Trees XML code

Listing A.1: Charge BT (Main Tree).

```
<root main_tree_to_execute = "MainTree" >
  <!-- MainTree ================================================================= -->

  <BehaviorTree ID="MainTree">
    <SequenceStar name="charge_sequence">
      <SubTree ID="PreApproachTree" />
      <SetBlackboard output_key="charger_id" value="1" />
      <SetBlackboard output_key="charger_pose" value="4;0;0" />
      <SubTree ID="ApproachChargerTree" input_id="charger_id" input_pose="charger_pose"/>
    </SequenceStar>
  </BehaviorTree>

</root>
```

Listing A.2: Charge BT (Sub Trees).

```
<!-- SubTrees ================================================================== -->

  <BehaviorTree ID="PreApproachTree">
    <SequenceStar name="pre_approach_sequence">
      <SetBlackboard output_key="approach_pose" value="2.4;0;0" />
      <Action ID="CallService" srv="/service_layer/get_robot_pose"
                                      output_pose="{robot_pose}" />
      <Fallback name="pre_approach_fallback">
        <Action ID="CheckPose" goal_pose="{approach_pose}"
                                      current_pose="{robot_pose}" />
        <Action ID="GoToPose" target_pose="{approach_pose}" />
      </Fallback>
    </SequenceStar>
  </BehaviorTree>

  <BehaviorTree ID="ApproachChargerTree">
    <SequenceStar name="approach_charger_sequence">
      <Action ID="CallService" srv="/service_layer/get_object_type"
                                        object_id="{input_id}"
                                        output_type="{object_type}"/>
      <Action ID="ApproachObject" object_id="{input_id}"
                                        object_type="{object_type}"
                                        object_pose="{input_pose}"/>
    </SequenceStar>
  </BehaviorTree>
```

Listing A.3: Simulation Test Case: Move and Check Pose BT code.

```
<root main_tree_to_execute = "MainTree" >

  <!-- MainTree ================================================================== -->

  <BehaviorTree ID="MainTree">
    <Sequence name="move_and_check_sequence">
      <SetBlackboard output_key="goal_pose" value="4;2;0" />
      <Action ID="GoToPose" target_pose="{goal_pose}" />
      <Action ID="CallService" srv="/service_layer/get_robot_pose" output_pose="{
          ↪ robot_pose}" />
      <Action ID="CheckPose" goal_pose="{goal_pose}" current_pose="{robot_pose}"
          ↪ goal_tolerance="0.1;0.1;0.09"/>
    </Sequence>
  </BehaviorTree>
</root>
```

Listing A.4: Simulation Test Case: Move and Check Pose (failure) BT code.

```
<root main_tree_to_execute = "MainTree" >

  <!-- MainTree ================================================================== -->

  <BehaviorTree ID="MainTree">
    <Fallback name="approach_pose_fallback">
      <Sequence name="move_and_check_sequence_failure">
        <SetBlackboard output_key="half_pose" value="2;4;0" />
        <Action ID="GoToPose" target_pose="{half_pose}" />
        <Action ID="CallService" srv="/service_layer/get_robot_pose" output_pose="{
            ↪ robot_pose}" />
        <Action ID="CheckPose" goal_pose="4;2;0" current_pose="{robot_pose}"
            ↪ goal_tolerance="0.1;0.1;0.09"/>
      </Sequence>
      <Sequence name="move_and_check_sequence_success">
        <SetBlackboard output_key="goal_pose" value="4;2;0" />
        <Action ID="GoToPose" target_pose="{goal_pose}" />
        <Action ID="CallService" srv="/service_layer/get_robot_pose" output_pose="{
            ↪ robot_pose}" />
        <Action ID="CheckPose" goal_pose="{goal_pose}" current_pose="{robot_pose}"
            ↪ goal_tolerance="0.2;0.2;0.09"/>
      </Sequence>
    </Fallback>
  </BehaviorTree>
</root>
```

Listing A.5: Simulation Test Case: Approach Charger BT code.

```
<root main_tree_to_execute = "MainTree" >

  <BehaviorTree ID="MainTree">
    <SequenceStar name="approach_sequence">
      <SetBlackboard output_key="container_pose" value="2;4.5;0" />
      <SetBlackboard output_key="object_id" value="31" />
      <Action ID="CallService" srv="/service_layer/get_object_type" object_id="{object_id}
          ↪ "
                                                          output_type="{object_type}"/>
      <Action ID="ApproachObject" object_id="{object_id}" object_type="{object_type}"
          ↪ object_pose="{container_pose}"/>
    </SequenceStar>
  </BehaviorTree>

</root>
```

Listing A.6: Simulation Test Case: Approach Container BT code.

```
<root main_tree_to_execute = "MainTree" >

  <BehaviorTree ID="MainTree">
    <SequenceStar name="approach_sequence">
      <SetBlackboard output_key="container_pose" value="2;4.5;0" />
      <SetBlackboard output_key="object_id" value="31" />
      <Action ID="CallService" srv="/service_layer/get_object_type" object_id="{object_id}
          ↪ "
                                                          output_type="{object_type}"/>
      <Action ID="ApproachObject" object_id="{object_id}" object_type="{object_type}"
          ↪ object_pose="{container_pose}"/>
    </SequenceStar>
  </BehaviorTree>

</root>
```

Listing A.7: Simulation Test Case: Approach Multiple Containers BT code (SubTrees).

```
  <!-- SubTrees ================================================================-->

  <BehaviorTree ID="MoveHomeTree">
    <SequenceStar name="move_home_sequence">
       <Action ID="GoToPose" target_pose="{input_pose}" />
    </SequenceStar>
  </BehaviorTree>

  <BehaviorTree ID="ApproachContainerTree">
    <SequenceStar name="approach_container_sequence">
      <Action ID="CallService" srv="/service_layer/get_object_type"
                               object_id="{input_id}"
                               output_type="{object_type}"/>
      <Action ID="ApproachObject" object_id="{input_id}"
                                  object_type="{object_type}"
                                  object_pose="{input_pose}"/>
    </SequenceStar>
  </BehaviorTree>
```

Listing A.8: Simulation Test Case: Approach Multiple Containers BT code (Main Tree).

```
<root main_tree_to_execute = "MainTree" >

  <!-- MainTree ============================================================================-->

  <BehaviorTree ID="MainTree">
    <SequenceStar name="approach_sequence">
      <SequenceStar name="approach_sequence_31">
        <!-- Variables -->
        <SetBlackboard output_key="container_id" value="31" />
        <SetBlackboard output_key="container_pose" value="4;5;0" />
        <!-- Actions -->
        <SubTree ID="ApproachContainerTree" input_id="container_id"
                                                input_pose="container_pose"/>
        <SubTree ID="MoveHomeTree" input_pose="home_pose"/>
      </SequenceStar>
      <SequenceStar name="approach_sequence_32">
        <!-- Variables -->
        <SetBlackboard output_key="container_id" value="32" />
        <SetBlackboard output_key="container_pose" value="4;2;0" />
        <SetBlackboard output_key="home_pose" value="0;0;0" />
        <!-- Actions -->
        <SubTree ID="ApproachContainerTree" input_id="container_id"
                                                input_pose="container_pose"/>
        <SubTree ID="MoveHomeTree" input_pose="home_pose"/>
      </SequenceStar>
      <SequenceStar name="approach_sequence_33">
        <!-- Variables -->
        <SetBlackboard output_key="container_id" value="33" />
        <SetBlackboard output_key="container_pose" value="4;-1.3;0" />
        <!-- Actions -->
        <SubTree ID="ApproachContainerTree" input_id="container_id"
                                                input_pose="container_pose"/>
        <SubTree ID="MoveHomeTree" input_pose="home_pose"/>
      </SequenceStar>
      <SequenceStar name="approach_sequence_34">
        <!-- Variables -->
        <SetBlackboard output_key="container_id" value="34" />
        <SetBlackboard output_key="container_pose" value="4;-5;0" />
        <!-- Actions -->
        <SubTree ID="ApproachContainerTree" input_id="container_id"
                                                input_pose="container_pose"/>
        <SubTree ID="MoveHomeTree" input_pose="home_pose"/>
      </SequenceStar>
    </SequenceStar>
  </BehaviorTree>

</root>
```

Listing A.9: Simulation Test Case: Approach Single Container configuration file.

```
containers:
  - id: 31
    dimensions:
      length : 1.2
      width : 1.0
    params:
      distance_from_object : 0.1
      distance_to_goal_tolerance : 0.1
      heading_tolerance : 0.01
      angular_velocity_approach : 0.3
      linear_velocity_approach : 0.3
      yaw_goal_tolerance : 0.01
```

Listing A.10: Simulation Test Case: Approach Multiple Containers configuration file.

```
containers:
  - id: 31
    dimensions:
      length : 1.2
      width : 1.0
    params:
      distance_from_object : 0.1
      distance_to_goal_tolerance : 0.1
      heading_tolerance : 0.01
      angular_velocity_approach : 0.3
      linear_velocity_approach : 0.3
      yaw_goal_tolerance : 0.01
  - id: 32
    dimensions:
      length : 1.2
      width : 1.0
    params:
      distance_from_object : 0.05
      distance_to_goal_tolerance : 0.05
      heading_tolerance : 0.01
      angular_velocity_approach : 0.3
      linear_velocity_approach : 0.3
      yaw_goal_tolerance : 0.01
  - id: 33
    dimensions:
      length : 1.2
      width : 1.0
    params:
      distance_from_object : 0.05
      distance_to_goal_tolerance : 0.05
      heading_tolerance : 0.01
      angular_velocity_approach : 0.3
      linear_velocity_approach : 0.3
      yaw_goal_tolerance : 0.01
  - id: 34
    dimensions:
      length : 1.2
      width : 1.0
    params:
      distance_from_object : 0.05
      distance_to_goal_tolerance : 0.05
      heading_tolerance : 0.01
      angular_velocity_approach : 0.3
      linear_velocity_approach : 0.3
      yaw_goal_tolerance : 0.01
```

B

Appendix 2 - Support Figures

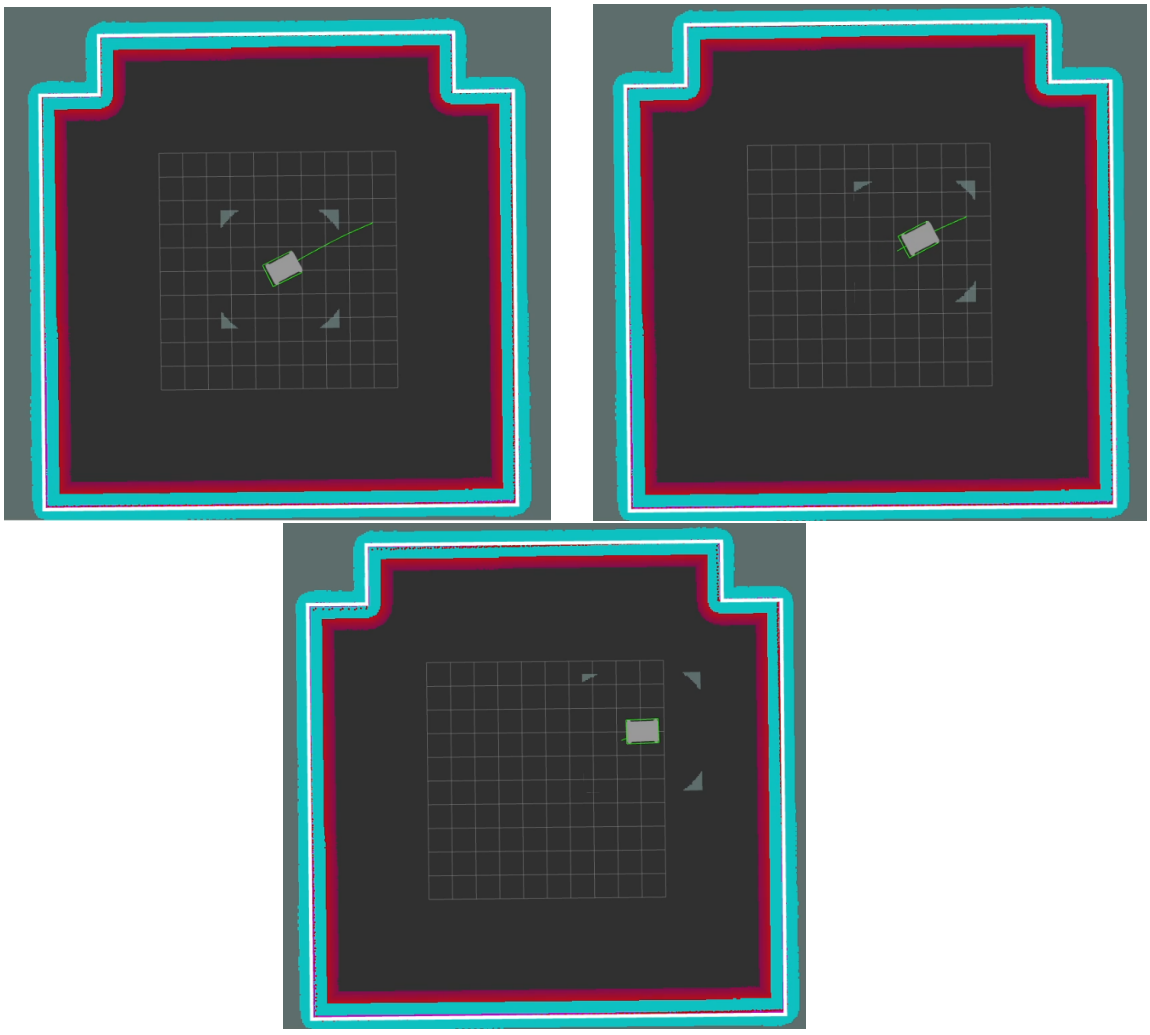


Figure B.1: Trajectory took by the AMR in the sequence (green line).

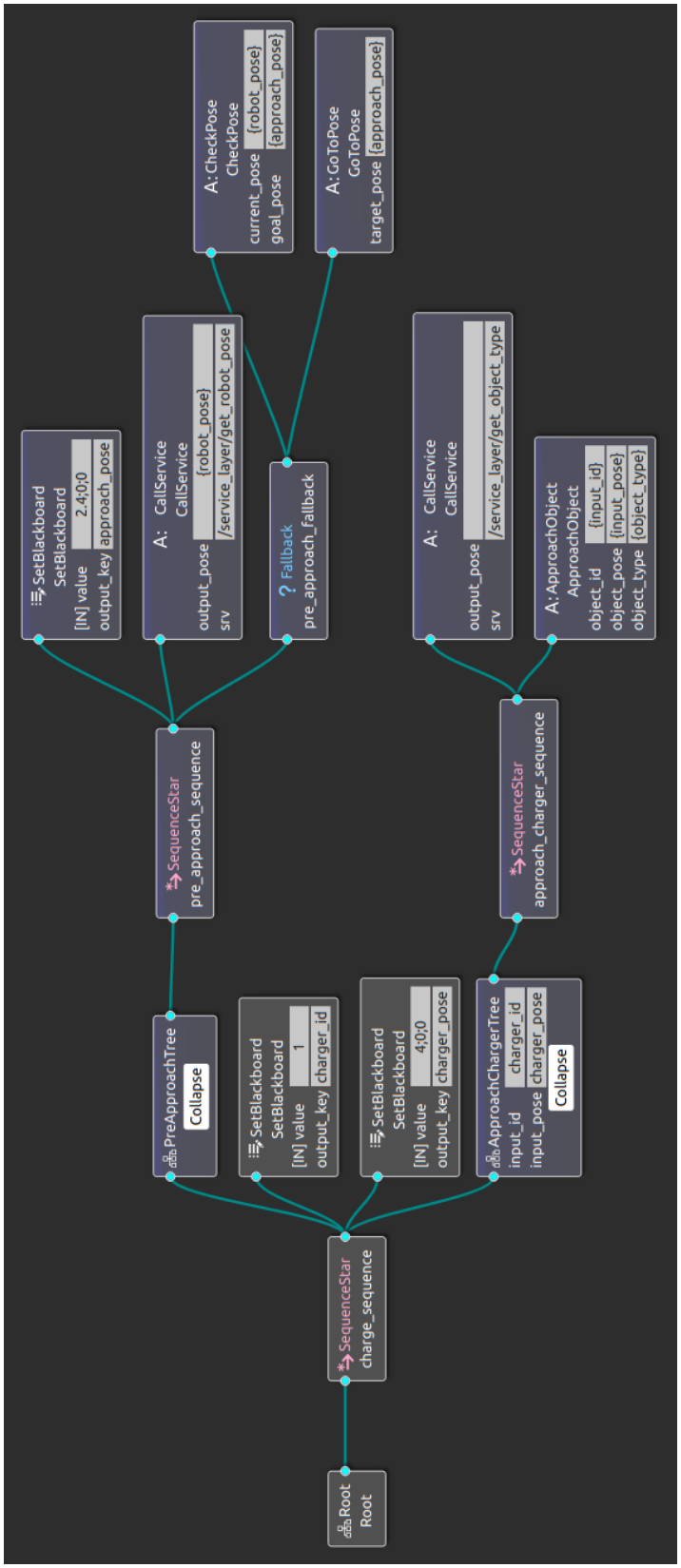


Figure B.2: Full Tree of the Charge behavior with the expanded subtrees.

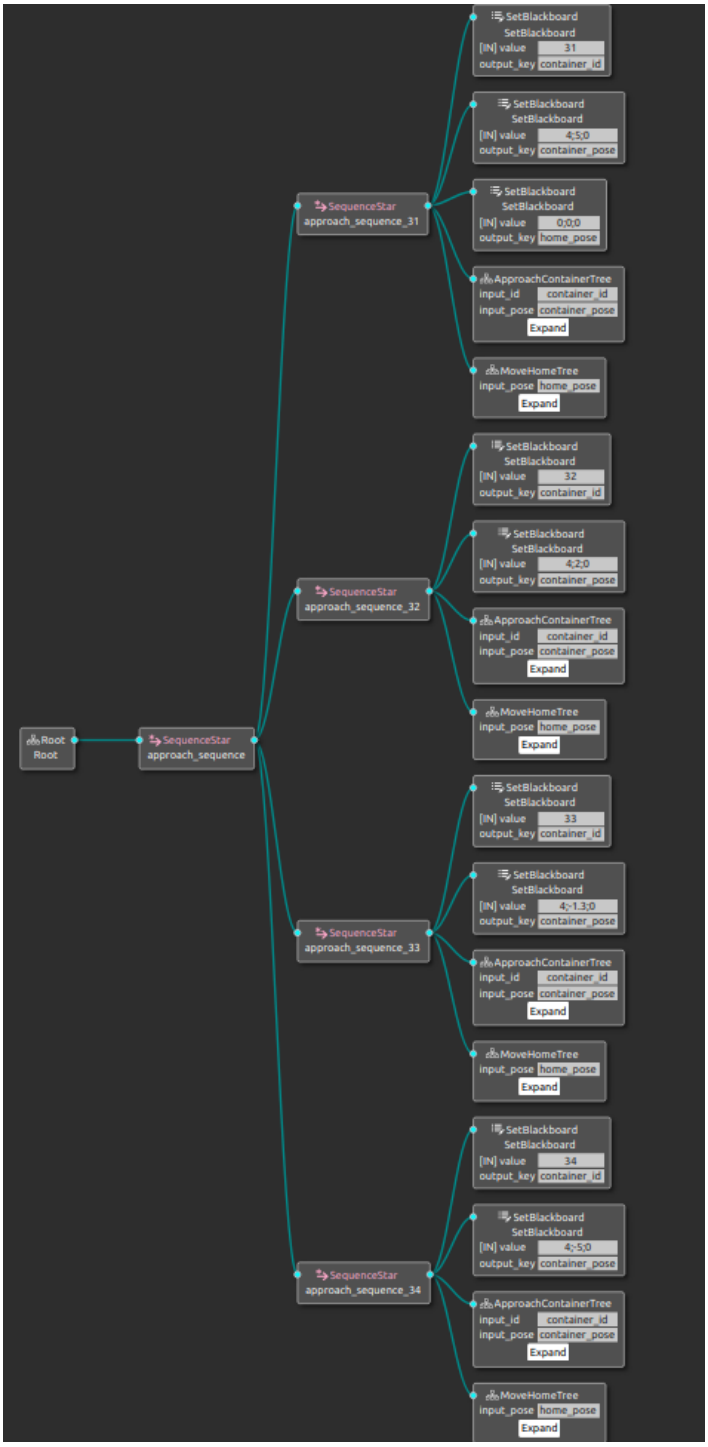


Figure B.3: Full BT of the approach multiple containers test case. The full process is a sequence star node (footnote of the first tree) with four nested sequences of the same type, one for each container. Values in the *SetBlackboard* node with '[IN]' before them, means that they will be stored inside the variable with the name in the output_key's field. The *ApproachContainerTree* and *MoveHomeTree* are *Sub Trees*, because they have an 'Expand' button, thus they can be expanded in more *action nodes*. Both trees have input variables so they can be accessed by the action nodes in them.

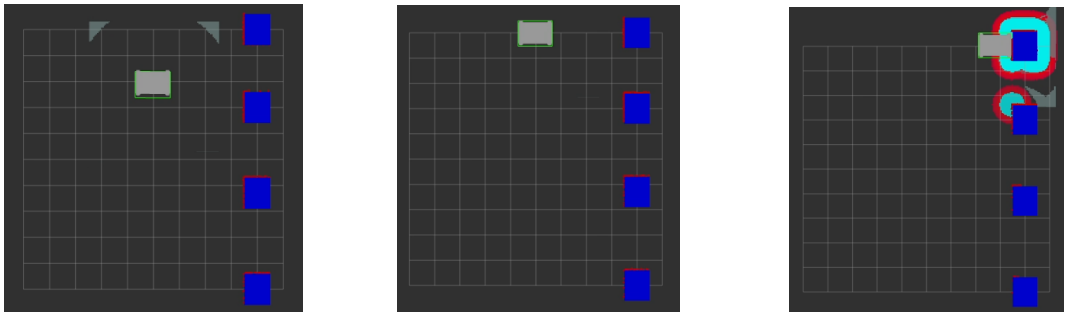


Figure B.4: Simulation of the AMR (grey shape) approaching container number thirty one (the first counting from upwards to downwards). The first figure (counting left to right) represents the alignment of the AMR with the container (dark blue square) in the y-axis, the second, the reduction of the distance in the x-axis and the third, the AMR aligned with the container thirty one. The coloured ellipses (light blue and red) are the inflation layer created by the costmap, the red dots around the containers are the laser scan beams and the green square around the AMR is the robot's footprint.

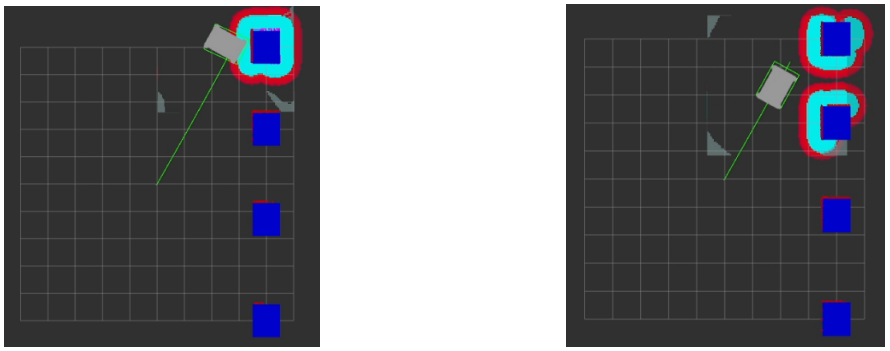


Figure B.5: AMR returning to the *Home Position*. The green line shows the trajectory followed by the AMR (grey shape) after approaching the container number 31 (dark blue square). The coloured ellipses (light blue and red) are the inflation layer created by the costmap, the red dots around the containers are the laserscan beams and the green square around the AMR is the robot's footprint.

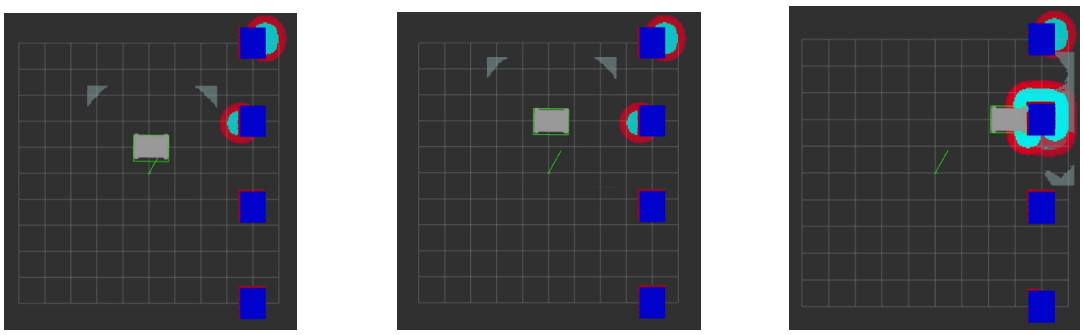


Figure B.6: Simulation of the AMR (grey shape) approaching container number thirty two (the second counting from upwards to downwards). The first figure (counting left to right) represents the alignment of the AMR with the container (dark blue square) in the y-axis, the second, the reduction of the distance in the x-axis and the third, the AMR aligned with the container thirty two. The coloured ellipses (light blue and red) are the inflation layer created by the costmap, the red dots around the containers are thye laserscan beams and the green square around the AMR is the robot's footprint.

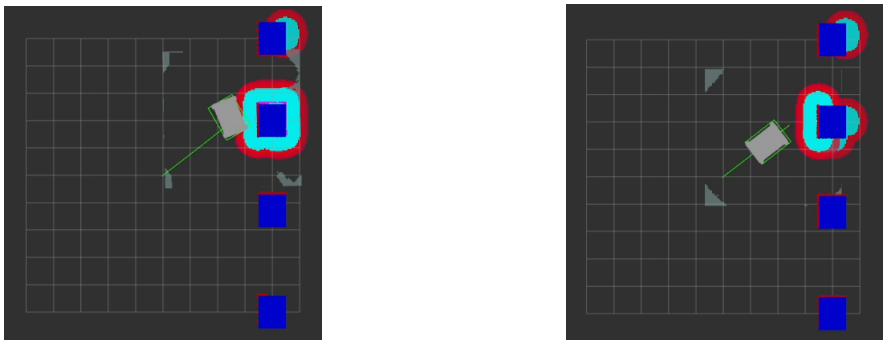


Figure B.7: AMR returning to the *Home Position*. The green line shows the trajectory followed by the AMR (grey shape) after approaching the container number 32 (dark blue square). The coloured ellipses (light blue and red) are the inflation layer created by the costmap, the red dots around the containers are the laserscan beams and the green square around the AMR is the robot's footprint.

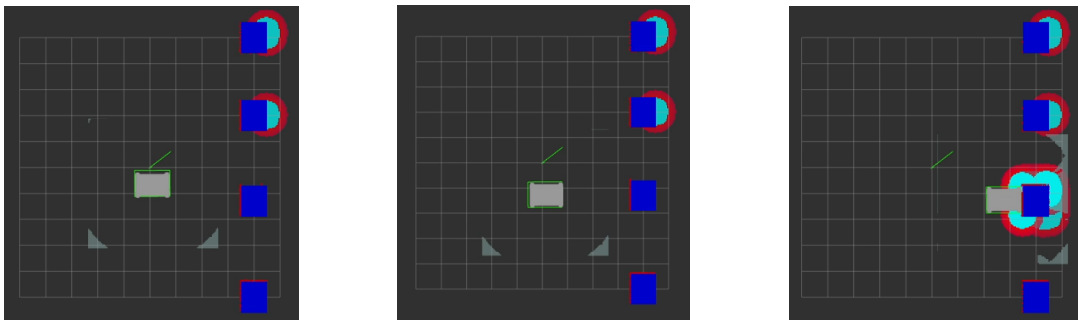


Figure B.8: Simulation of the AMR (grey shape) approaching container number thirty three (the second counting from upwards to downwards). The first figure (counting left to right) represents the alignment of the AMR with the container (dark blue square) in the y-axis, the second, the reduction of the distance in the x-axis and the third, the AMR aligned with the container thirty three. The coloured ellipses (light blue and red) are the inflation layer created by the costmap, the red dots around the containers are thye laserscan beams and the green square around the AMR is the robot's footprint.

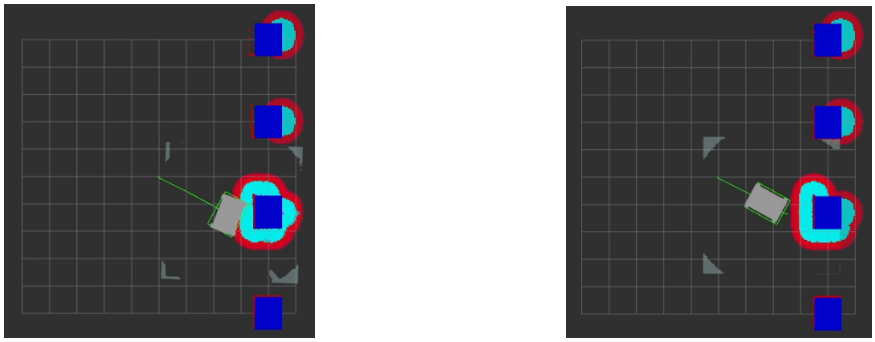


Figure B.9: AMR returning to the *Home Position*. The green line shows the trajectory followed by the AMR (grey shape) after approaching the container number 33 (dark blue square). The coloured ellipses (light blue and red) are the inflation layer created by the costmap, the red dots around the containers are the laserscan beams and the green square around the AMR is the robot's footprint.

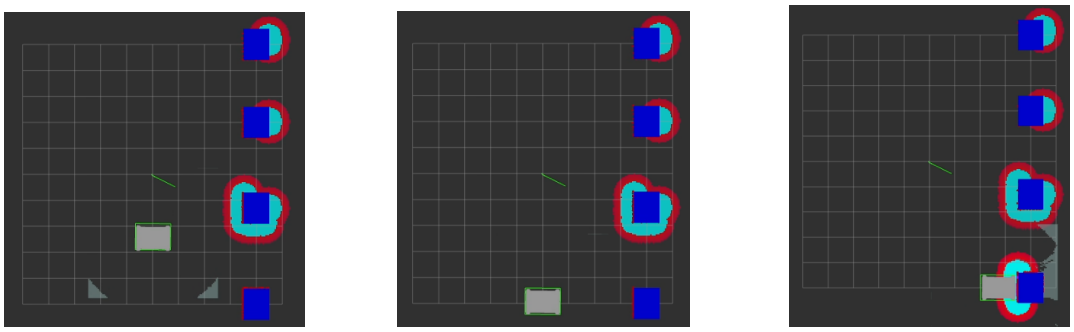


Figure B.10: Simulation of the AMR (grey shape) approaching container number thirty four (the second counting from upwards to downwards). The first figure (counting left to right) represents the alignment of the AMR with the container (dark blue square) in the y-axis, the second, the reduction of the distance in the x-axis and the third, the AMR aligned with the container thirty four. The coloured ellipses (light blue and red) are the inflation layer created by the costmap, the red dots around the containers are the laserscan beams and the green square around the AMR is the robot's footprint.

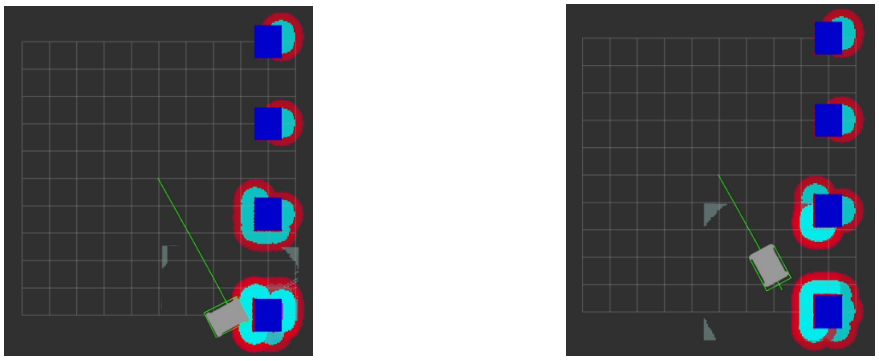


Figure B.11: AMR returning to the *Home Position*. The green line shows the trajectory followed by the AMR (grey shape) after approaching the container number 34 (dark blue square). The coloured ellipses (light blue and red) are the inflation layer created by the costmap, the red dots around the containers are the laserscan beams and the green square around the AMR is the robot's footprint.

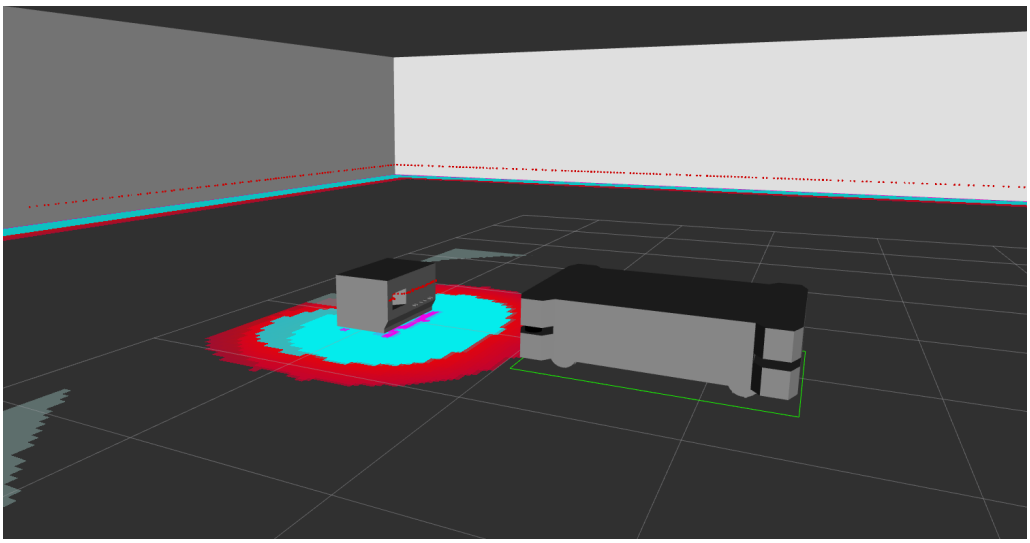


Figure B.12: AMR approaching the charger (grey parallelepiped). The green rectangle around the AMR is the robot's footprint. The red dots in the front face of the charger and the walls are laserscan beams colliding with them. The coloured ellipses on the ground and lines near the walls represent the costmap's inflation layer which is created around every obstacle. The colours represent the increase of the trajectory weight around the charger and close to the walls.

Figure B.13: AMR connected to the charger (grey parallelepiped). On the left side of the figure, inside the red ellipse, it it is mentioned two poses (position and orientation). The first pose is the difference between the AMR center projected on the ground, and the charger's pins. The second (relative), it's the pose given by the robot's odometry (position the robot thinks it is on the map). The coloured ellipses on the ground represent the costmap's inflation layer which is created around every obstacle. The colours represent the increase of the trajectory weight around the charger.

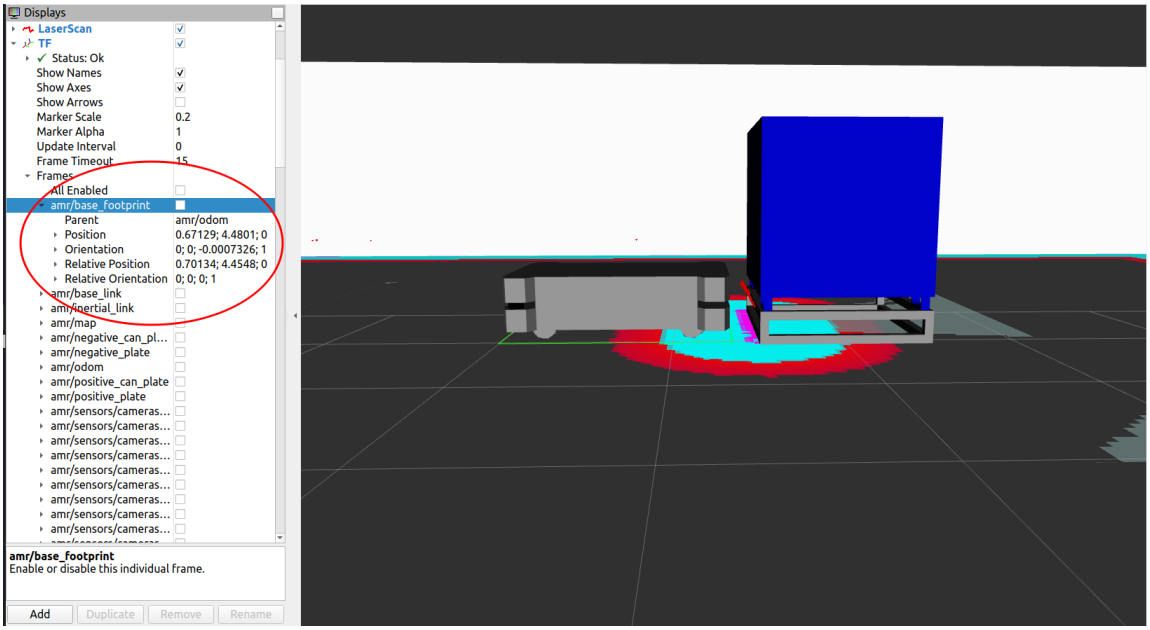


Figure B.14: AMR final pose in front of the container (blue and grey object). On the left side of the figure, inside the red ellipse, it it is mentioned two poses (position and orientation). The first, it's the AMR's pose in the map. The relative pose it's the position given by the robot's odometry (position the robot thinks it is on the map). The coloured ellipses on the ground represent the costmap's inflation layer which is created around every obstacle. The colours represent the increase of the trajectory weight around the charger.