

Behavioral Planning Approaches To Improve Autonomous Behaviour During Navigation for Mobile Robots Running ROS

Bachelorarbeit

Name des Studiengangs
Ingenieurinformatik

Fachbereich 2

vorgelegt von
Lukas Evers

Datum: 28.09.2022

Erstgutachter: Prof. Dr. Frank Burghardt
Zweitgutachter: Dr.-Ing. Ahmed Hussein

Contents

1	Introduction	2
2	State of the Art	4
2.1	Automated and Autonomous Vehicles	4
2.2	Autonomous Driving Navigation Architectures	4
2.3	Behavior Types	6
2.3.1	Reactive	6
2.3.2	Deliberative	7
2.3.3	Hybrid	7
2.4	Behavior Planning Approaches	8
2.4.1	Finite State Machines	9
2.4.2	Behavior Trees	10
2.4.3	Partially Observable Markov Decision Processes	11
3	Concept	13
3.1	Current State	13
3.1.1	ROS	13
3.1.2	Navigation2	14
3.1.3	Turtlebot3	16
3.2	Requirements	17
3.3	System Solution	17
3.4	Software and Simulation	19
3.4.1	Behaviortree.CPP	19
3.4.2	Groot	21
3.4.3	Gazebo Simulation Environment	21
3.4.4	OS and Software Versions	22
4	Implementation	24
4.1	System Architecture	24
4.2	Data Backup	25
4.3	Command Velocity Decision Gate	26
4.4	Behavior Tree Structure	26
4.4.1	BT System Supervision	26
4.4.2	BT Advanced Behaviors	27
4.5	System Support Tools	30

4.5.1	Gazebo Sensor Drivers	30
4.5.2	Custom ROS Interfaces	31
5	Evaluation	35
5.1	Scenarios	36
5.2	Results	37
6	Conclusion	40
7	Outlook	41

List of Figures

2.1	SAE Levels Of Driving Automation [2]	5
2.2	Common Autonomous Driving Architecture [4] [5]	6
2.3	Hybrid Planning Architecture [10]	8
2.4	Example of a finite state machine transition diagram [11]	9
2.5	Example of a behavior tree [17]	11
2.6	Example of a fully observable Markov Decision Process [19]	12
3.1	Navigation2 Architecture [24]	14
3.2	Navigation2 Behavior Tree	15
3.3	Turtlebot3 model in Gazebo simulator	16
3.4	Use Case Diagram	20
3.5	System Block Diagram	21
3.6	Live monitoring of behavior trees with Groot	23
4.1	Component Diagram of System Architecture	24
4.2	Top Level Behavior Tree Structure	26
4.3	Examiner pattern for the system supervisor component [25]	27
4.4	Lidar Fallback	28
4.5	Collision Fallback Behavior	28
4.6	Activity Diagram for the Collision Behavior	32
4.7	Battery Fallback Behavior	33
4.8	Path Planning Fallback Behavior	33
4.9	Graph for the Sensor Driver nodes and topics	34
5.1	Apartment Environment in Gazebo	35
5.2	Enclosed Environment in Gazebo	36

List of Tables

2.1	The five types of nodes [17]	10
3.1	ROS Communication Options	13
3.2	Turtlebot3 Specifications	17
3.3	Functional Requirements	18
3.4	Non-Functional Requirements	19
3.5	Control Nodes in BT.CPP	21
3.6	Decorator Nodes in BT.CPP	22
3.7	Used Software Versions	23
4.1	Types of saved data	25
4.2	Implemented Custom Services	31
5.1	Sensor Scenarios	38
5.2	Behavior Scenarios	39
5.3	Results	39

Abstract

This thesis explores ways to improve the autonomy of mobile robots running the Robot Operating System (ROS) with behavioral planning approaches. Autonomous mobile robots often require close human supervision or intervention during their operation.

A system supervision system, a sensor data storage and a behavior tree were designed and implemented. The system was tested on a simulated robot in the Gazebo simulator. Behaviors to handle sensor failures, collisions, unreachable goals, and low battery state.

The different test scenarios were artificially induced in a standardized manner to trigger the behaviors to react to the scenarios. The results show an increase in the robot's autonomy and robustness to failures.

The results need to be tested and verified on a real robot, but the increase in the robot's autonomy in the simulation tests are promising an equally positive result. Behavior trees to control and improve the behavior of robots offer the ability to effortlessly add new behaviors to an existing system. This thesis shows a comprehensive way how behavior trees can be implemented to an existing ROS2 system.

Chapter 1

Introduction

Mobile robots are becoming more widespread in many domains. They are marketed as autonomous mobile robots, but their autonomy is often limited to particular environments and requires frequent human help to function correctly. This reliance on a human operator hardly qualifies the robot to be labeled autonomous.

One of the quickest and most popular ways to build and program a robot is to use the Robot Operating System (ROS). ROS offers many tools and functionalities to create and program robots, but a standard robot needs human supervision during the operation to ensure proper function and safety. There are ROS packages to safely navigate from point A to point B, but currently, typical robots cannot react to unforeseen events. The robot needs to be able to navigate in an uncertain environment to guarantee the safety of the robot itself and all other actors in its environment. These can be either other robots or humans.

In theory, the robot can perform fully autonomous navigation, including mapping and localization. However, when something unexpected happens, the robot's autonomy can not be guaranteed any more. A default robot cannot represent all possible fail states and does not detect failures on a systematic level but only inside its navigation-related subsystems. Moreover, when failures inside these subsystems are discovered, the options to handle these problems are minimal and often do not deal with the problems in an autonomous way. The reliance on external problem-solving decreases the robot's usefulness when tasked with tangible goals. Therefore, this thesis centers around the research question of how behavior planning can increase the robustness and autonomy of a robot running ROS2. Behavior planning enables the robot to react to failures and problems of the system and can decide alternative courses of action to mitigate risks and finish the given tasks.

This thesis will explore possibilities for improving current systems by adding a dedicated behavior planning component. Furthermore, the thesis will provide an implementation of exemplary behaviors and provide a system architecture to incorporate behavior planning in other robots, too.

The desired outcomes of this implemented software will be an increase in robustness and a decrease in required human actions during several scenarios. In these scenarios, failures

and problems are artificially induced to test the robot's abilities to behave autonomously.

Chapter 2

State of the Art

2.1 Automated and Autonomous Vehicles

This thesis looks for a way to improve the autonomy of mobile robots. In order to define an improvement in autonomous behavior, one needs to explore the definitions of the terms "automated" and "autonomy". The "Expertenkommission Forschung und Innovation" (EFI) defines the term autonomy in the context of robotics as a system that can act without human instructions and still solve complex tasks, make decisions, learn independently as well as react to unforeseen circumstances [1]. The definition specifies the needed requirements to make a robot fully autonomous. The automotive industry defines vehicle autonomy based on the human driver's need for supervision and possible intervention when executing complex driving tasks [2]. Figure 2.1 depicts the different levels of autonomy of passenger cars on a scale between no automation to completely autonomous. Both definitions rely on human supervision and guidance as the central part of what hinders a system or vehicle from becoming fully autonomous. By decreasing the instances of human intervention during the operation of a robot, one can increase the automation level towards full autonomy. However, this has to be done by equipping the robot with robust and context-driven decision-making and planning capabilities. Otherwise, a robot that never needs a human to operate could easily be created. Nevertheless, this robot could not be considered autonomous as it would not be able to solve complex tasks and make intelligent decisions.

2.2 Autonomous Driving Navigation Architectures

To better understand how behavior planning influences a robot's autonomy, one can look at the latest and most used software architectures for autonomous driving on a functional level. Most of the current autonomous driving architectures implement a hierarchical structure that follows the "Sense - Think - Act" paradigm [3]. As shown in figure 2.2, the sensory inputs, usually from multiple sensors, get processed to create an environment representation. Then the system calculates how to get from its current position to the destination and creates a path. A second planning calculation is triggered to compute the motion commands with the system's constraints (e.g., size, turning radius). These motion commands then get converted to control the motors.

SAE LEVEL 0™	SAE LEVEL 1™	SAE LEVEL 2™	SAE LEVEL 3™	SAE LEVEL 4™	SAE LEVEL 5™
What does the human in the driver's seat have to do?	You are driving whenever these driver support features are engaged – even if your feet are off the pedals and you are not steering	You must constantly supervise these support features; you must steer, brake or accelerate as needed to maintain safety	You are not driving when these automated driving features are engaged – even if you are seated in “the driver’s seat”	When the feature requests, you must drive	These automated driving features will not require you to take over driving
What do these features do?	These features are limited to providing warnings and momentary assistance	These features provide steering OR brake/acceleration support to the driver	These features provide steering AND brake/acceleration support to the driver	These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met	This feature can drive the vehicle under all conditions

Copyright © 2021 SAE International.

	These are driver support features			These are automated driving features	
What do these features do?	These features are limited to providing warnings and momentary assistance	These features provide steering OR brake/acceleration support to the driver	These features provide steering AND brake/acceleration support to the driver	These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met	This feature can drive the vehicle under all conditions
Example Features	<ul style="list-style-type: none"> • automatic emergency braking • blind spot warning • lane departure warning 	<ul style="list-style-type: none"> • lane centering OR • adaptive cruise control 	<ul style="list-style-type: none"> • lane centering AND • adaptive cruise control at the same time 	<ul style="list-style-type: none"> • traffic jam chauffeur 	<ul style="list-style-type: none"> • local driverless taxi • pedals/steering wheel may or may not be installed

Figure 2.1: SAE Levels Of Driving Automation [2]

The planning sequence can be further divided into global, behavioral, and local planning. Global planning, also named route planning, is responsible for outputting a path from start to goal. Global planning is comparable to a person using Google Maps to find the shortest or fastest path to a faraway city. In this analogy, the behavior planner is responsible for following the traffic rules on the trip to the destination, e.g., stopping at red lights, giving the right of way to other vehicles, and following speed limits. The local planning, also named motion planning, takes the input from the behavior layer and is tasked to let the vehicle drive according to the executed behavior so that it stays in its lane and stops at a red light at the correct position[6].

Due to safety considerations, many autonomous driving architectures implement an additional system supervision layer that monitors the execution of the other system components. This supervisor checks the health of all software components. If one or more components fail to pass the health check, the supervisor is responsible for ensuring that the system does not continue the regular driving routine. The supervisor triggers measures to restore the standard functionality or, if that is not possible, bring the robot into a safe state [7]. Considering that with these architectures, vehicles can achieve an SAE level of autonomy up to level 4 [8], a good starting point to increase autonomy in a robotic system would be to ensure that all of the functional modules in figure xx are implemented and available. On this basis, higher autonomy levels are achievable by expanding the behavior planning module

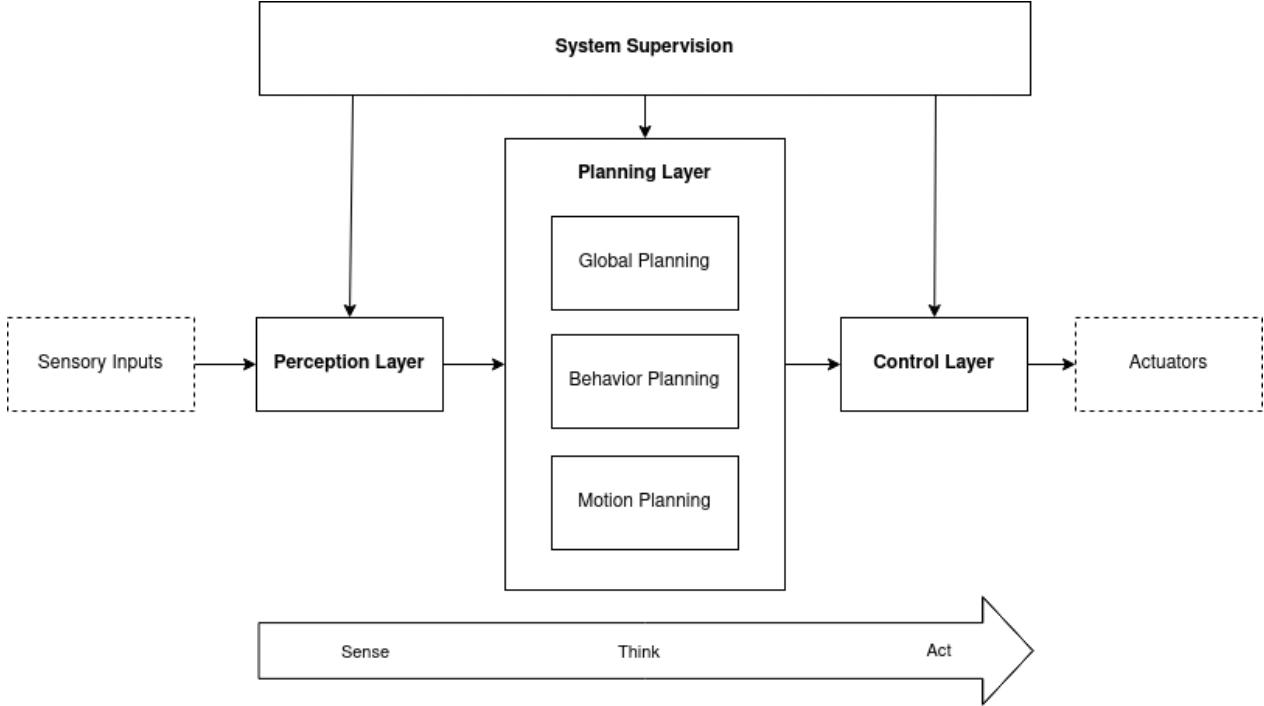


Figure 2.2: Common Autonomous Driving Architecture [4] [5]

inside the planning layer.

2.3 Behavior Types

The behavior planning module contains a set of behaviors to operate in an environment. The module's task is then to choose the best behavior to execute. In this sense, behavior is defined as a mapping of sensory inputs to a pattern of actions that carry out a specific task [3]. A good behavior planning approach provides the robot with acceptable behaviors for every possible scenario the robot is operating in. Different scenarios pose different challenges for the behavior planning module, which has to decide on the best behavior option to achieve a higher level goal.

2.3.1 Reactive

During the emergence of behavior-based robots in the 1960s, the first type of behaviors implemented were simple reactive behaviors. This behavior type maps a sensory input directly to motor commands. In human behavior, this behavior resembles reflexes, like tapping on the kneecap, which unwillingly results in motion in the knee joint. This behavior pattern is not following the typical "Sense, Think, Act" loop but shortcuts directly from sensing to acting [9]. Reactive behaviors can be chained together to achieve more advanced and goal-driven behaviors.

Despite the possibility of more complex robot behaviors, sequential behaviors remain on the level of reactive behaviors due to the lack of a planning and decision-making cycle during their execution. The computational load of reactive behaviors is low and thereby fast as no decision and planning process is taking place, which makes them suited in scenarios where real-time safety is of concern. This property makes this behavior useful for system supervisors, where immediate reactions with low latency are sometimes required to ensure system safety. However, a system with a reactive behavior planning approach will always fail to meet the requirements for higher levels of autonomy due to the reasons discussed in previous chapters. This systematic lack of autonomy does not mean reactive behaviors can not be part of highly autonomous systems. Their quick response time to sensor inputs makes them valuable in improving vehicle safety and reliability.

2.3.2 Deliberative

Reactive behaviors suffer the drawback that they are not suitable to enable complex behaviors. In addition, creating sequences of reactive behaviors that produce the target behavior is a complicated task [3]. Behaviors that involve planning and decision-making aspects are defined as deliberative. Unlike reactive behaviors, deliberative behaviors follow the Sense, Think, Act paradigm, and they are not mapping sensory inputs directly into motor commands. Instead, deliberative-type behaviors can decide the best course of action before they act. Using deliberative behaviors in the planning module enables a robot to meet the definition of autonomy, e.g., making decisions and reacting to unforeseen circumstances.

Concluding, the focus should be on creating deliberative behaviors to improve autonomy. A behavior planner with a deliberative approach decides based on current sensor data and previously processed data. By incorporating older sensor information into the decision-making process, the deliberative behavior planner can predict the changes in the environment. Using older data points allows the planner to proactively change the motion planning commands to better adapt to the environment. For example, a motion prediction of obstacles in highly dynamic environments would improve motion planning considerably. The behavior planner can adjust the motion planning with improved information on how dynamic obstacles interact with the robot's planned path in the future. A purely reactive planner could never determine if an obstacle is destined to intersect the robot's path in the future and would constantly reroute to accomplish the given goal. On the other hand, a deliberative planner can determine if the best course of action is to stay on the current path as the dynamic obstacle has already moved away by the time the robot reaches the intersection point. Other actions in the scenario are possibly slowing down or speeding up briefly to avoid an obstacle and staying on the current path, which is calculated to be quicker than rerouting and taking a longer path. Generally, these cognitively more complex behaviors mimic human skills and thus make a system more autonomous.

2.3.3 Hybrid

One can combine reactive and deliberative behaviors in a hybrid model to create reliable and intelligent systems. This behavior planning module can quickly react to incoming sensor

data and still exhibit high levels of automation.

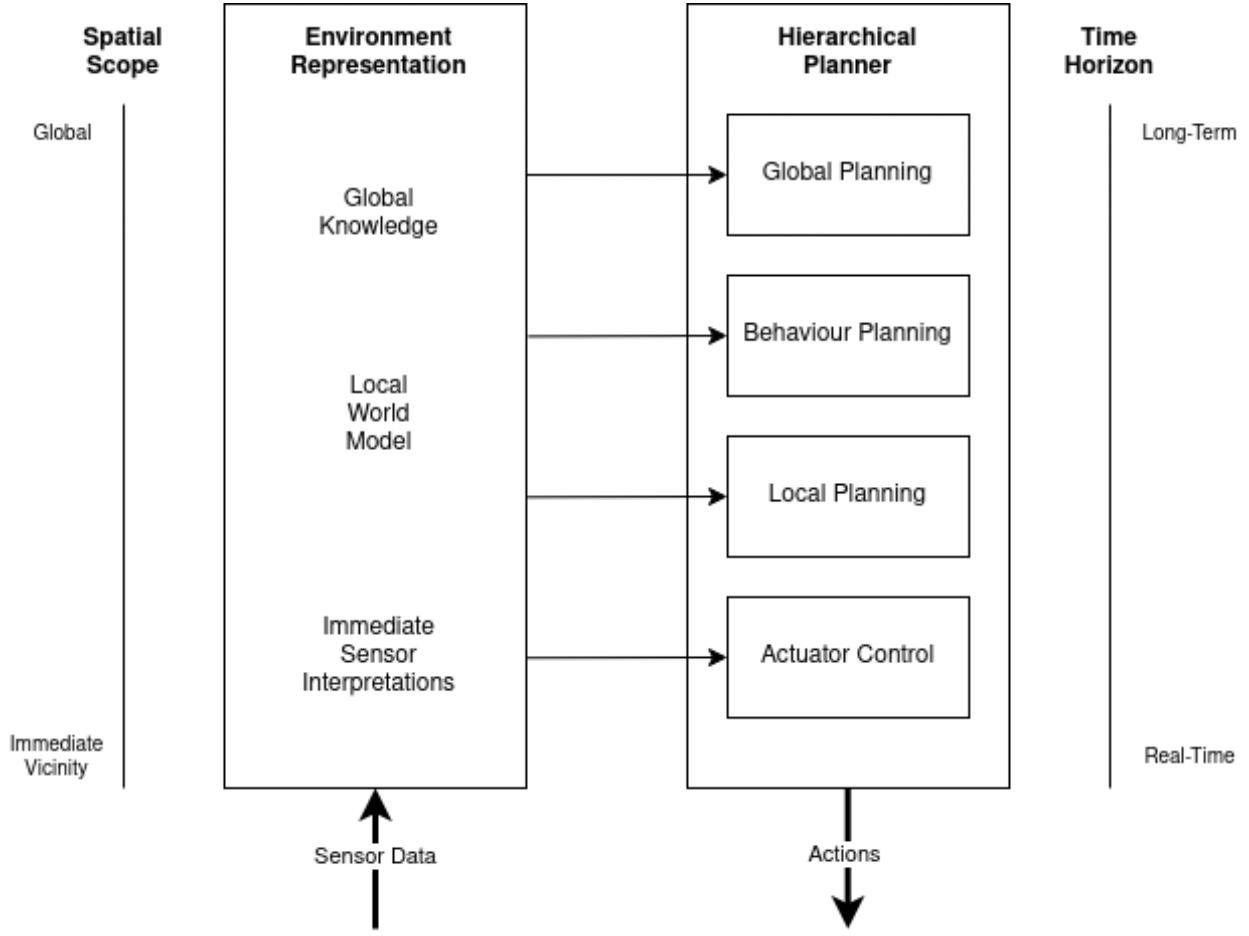


Figure 2.3: Hybrid Planning Architecture [10]

Figure 2.3 shows the areas where reactive and deliberative behaviors have advantages and limitations. Deliberative planning does possess limitations during the execution of generated plans as the time horizon is long-term and not able to react to fast changes in the environment. When the system is not addressing these points, deliberative robots are very focused on a small problem domain and are not robust [10]. The incorporation of reactive behaviors into behavior planning combats this problem. In this multi-layer approach, the higher level, more deliberative planners can override the reactive planners to allow the system to be more flexible and adaptable while maintaining fast reaction times. This architecture leads to more robust robots deployed in a wider variety of environments, thus leading to higher levels of autonomy.

2.4 Behavior Planning Approaches

Robots that use a hybrid, hierarchical behavior planning approach need a system that decides on a high level which behaviors are to be executed. That system allows overriding simple

reactive behaviors with more complex, deliberative behaviors when the whole system benefits from them.

The different solutions for behavior execution systems all share that they use states a robot can be in. Different strategies and behaviors are executed during the system's runtime based on a robot's state and additional information about the environment.

In the following sections, different approaches for behavior planning are presented and compared.

2.4.1 Finite State Machines

Finite State Machines (FSM) are commonly used in the autonomous driving domain. They describe behavior in the form of states, which trigger the execution of actions. A finite state machine is defined by a list of possible states S , a starting state s_0 , a set of state transitions δ , a set of final states F , and the input alphabet σ . At any given time, only a single state is selected, and its containing actions are executed.

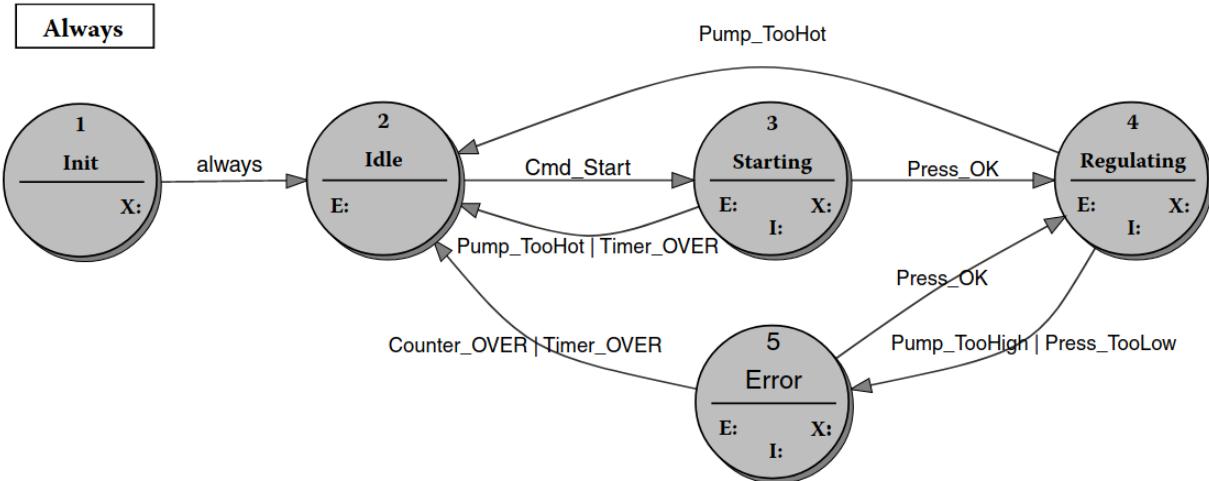


Figure 2.4: Example of a finite state machine transition diagram [11]

Figure 2.4 depicts an example state machine with different states and exemplary transitions between them. Inside of the states the letters indicate the existence of actions. E stands for entry, I for Input and X for Exit. The arrows between the states illustrate the possible transition and transition conditions for every state. A state transition diagram is an excellent way to understand the system behavior but can not show the details of the modeled behavior, like the actions [11].

Creating small state machines can easily and quickly be accomplished using various feature-rich, and performant libraries [12]. Finite state machines are often used to model reactive behavior comprehensively, and sequential behaviors in autonomous driving functions [13]. These finite state machines use nested state machines inside another state machine. The

nested state machines reduce the complexity of the state machine as the transitions do not need to be modeled because a nested state machine's start and final state are treated as just one state inside the bigger state machine. Despite this possibility to simplify state machines, as a modeled behavior system grows larger to accommodate more complex behaviors into the state machine, the effort to expand and maintain the state machine multiplies rapidly. The higher effort is due to the modeling of the transitions, transition conditions, and transition events growing with each new state that gets introduced into the state machine as existing states need to be checked and updated accordingly [14].

The execution time of finite state machines can be fast enough that they are used to control the motors of a bipedal robot to enable the robot to balance and walk despite unexpected height variations between steps [15]. This property makes state machines very well suited for fast, reactive type behaviors and still allows a hierarchical approach to behavior planning.

2.4.2 Behavior Trees

Behavior Trees (BT) are another way to model and control the behavior of autonomous systems. Behavior Trees first found considerable acceptance in the computer game industry, where they are mainly used to model artificial intelligence for non-player characters [16]. Every tree has one root and many children, parent, and leaf nodes. Leaf nodes are also called execution nodes, while non-leaf nodes are control or control flow nodes.

Table 2.1: The five types of nodes [17]

Node type	Symbol	Succeeds	Fails	Running
Sequence	->	If all children succeed	If one child fails	If one child returns running
Fallback	?	If one child succeeds	If all children fail	If one child returns running
Parallel	-> ->	If $\geq M$ children succeed	If $> N-M$ children fail	else
Action	shaded box	Upon completion	When impossible to complete	During completion
Condition	white oval	If true	If false	Never

The execution of a behavior tree is done by ticking the tree's root node. This signal then travels down to the child node of the root node, where either control nodes are ticked, or action nodes are executed. Nodes return either "Success", "Running" or "Failure" which influences how the tick signal gets processed by the rest of the behavior tree. Control nodes can be of the type "Sequence", "Fallback" or "Condition". The condition node can not have child nodes and can only return success or failure. A sequence node can be compared to a logical "and" condition. This property of the sequence node means that once a sequence gets ticked from the parent node, it will send the tick signal to every child node as long as

they return "Success" or "Running" and only return "Success" itself when every child node was successfully ticked. If any children nodes return "Failure", the sequence node will stop ticking the remaining children and return "Failure". On the other hand, the fallback node is an equivalent of a logical "or" condition. The fallback node ticks its children nodes as long as they return "Failure" or "Running" and will return "Success" if one of the ticked nodes returned "success". It will only return "Failure" if all children nodes returned "Success". The primary node types and their control flow modification are listed in table 2.1. Figure 2.5 depicts an example of a BT with multiple levels and action nodes. The nodes with the arrow (\rightarrow) symbol are sequence nodes, and the question mark (?) symbol denotes fallback nodes. The condition nodes are depicted as white ovals, and the action nodes are represented as gray rectangles in the behavior tree example.

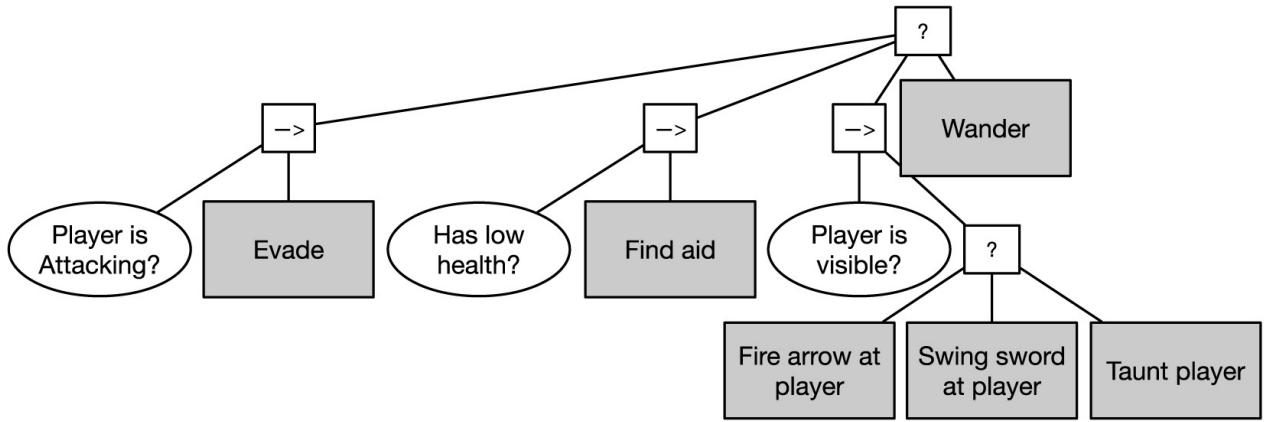


Figure 2.5: Example of a behavior tree [17]

One of the core differences to the Finite State Machine is that transitions between states are not distributed across all the states. In contrast, they are organized in a hierarchical tree, where the leaves represent the states [17]. FSM and BT can produce the same behavior in robots. However, the fundamental shift in how the two systems are created leads to significant advantages in the modularity, synthesis, and analysis of the systems at hand. These effects are more significant with an increase in the size of the system. Behavior trees offer much more flexibility when creating an advanced behavior layer for an autonomous system.

2.4.3 Partially Observable Markov Decision Processes

Another approach to behavior planning is the Markov Decision Process(MDP) and the further advanced Partially Observable Markov Decision Process (POMDPs). MDPs share structural features with Finite State Machines, but the behavior planning and state transitions are stochastic and not deterministic in contrast to FSMs. An MDP can be described as a tuple $\langle S, D, A, T, R \rangle$. Like a finite state machine, the MDP has a set of possible states S and a set of possible transitions T between them. The transitions contain a probability of execution and must add up to 1.0 for each associated state. An MDP adds a set of finite actions A and rewards R to the system. Additionally, MDPs possess a set of discrete time steps, in this context often called epochs, D , which can be infinitely long. The guiding principle of MDPs

is to maximize the expected reward by deciding on the best action. This policy Π is what guides the behavior of the MDP [18].

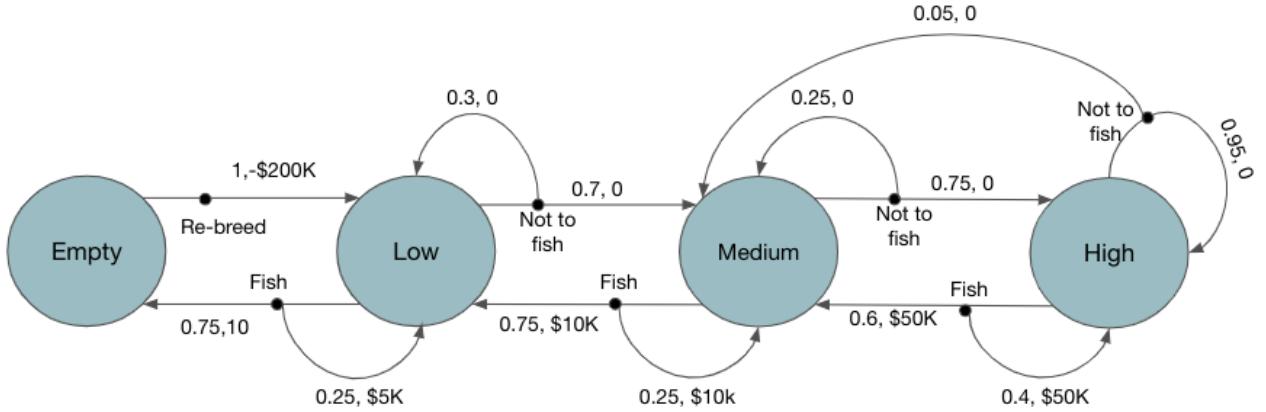


Figure 2.6: Example of a fully observable Markov Decision Process [19]

Figure 2.6 shows an exemplary MDP for determining a fisherman's best course of action. The states are the population sizes of available fish, and the only available actions are to either fish or not to fish. The actions possess different rewards depending on the current state of the system [20].

The difference between an MDP and a POMDP is that the system's current state can be unknown, and the system can make observations and estimate the state to a degree. The POMDP is defined as a tuple $\langle S, D, A, T, R, Z, O \rangle$, in which the MDP tuple gets extended by Z and O . Z is the observation, and O is the probability function of seeing Z in a given state S .

POMDPs are well suited for applications in unknown environments because the system can make decisions when the perception information is imperfect, or the environment is not fully observable. The decision process is probabilistic and is more adaptable when confronted with unknown scenarios. This flexibility comes at the cost of determinism. The system's behavior is not guaranteed to be the same whenever it is subjected to the same sensory input [21].

Chapter 3

Concept

This chapter briefly presents the current state of mobile navigation with ROS and its limitations. This chapter aims to identify the critical areas in which standard mobile robots running ROS lack robustness and autonomy during autonomous mobile navigation. According to these critical areas, requirements are derived from them and prioritized with a risk analysis. This chapter proposes how behavior planning can improve the current default systems to meet these requirements.

3.1 Current State

3.1.1 ROS

The Robot Operating System (ROS) is an open-source middleware for creating robot applications. It is not an operating system (OS) since it needs an underlying OS, mainly Linux-based, to work. ROS can handle the communication between different programs (called nodes) with standardized interfaces and network protocols. Nodes can communicate via topics, services, and actions, which differ in their application domain (compare table 3.1).

Table 3.1: ROS Communication Options

Name	Communication Pattern	Area of Use	Cardinality
Topics	Publisher/ Subscriber	Continuous stream of data e.g. sensor data	n:m
Services	Request/ Response	Get specific data only once	One service, many clients
Actions	Request/ Response and Publish/ Subscribe	Trigger executing of asynchronous goal-driven processes and receive updates	One action, many clients

This design choice makes robotic applications built with ROS highly reusable and interchangeable, allowing developers to integrate foreign libraries more easily into their applications. Due to the availability of many high-quality open-source libraries for many different

use cases, ROS has become the leading methodology for creating robotic applications in the research environment [22].

With ROS2, the framework received fundamental changes and updates to its architecture and design to gain more acceptance and use in the industry. These changes allow for real-time safety, simpler certification, and security when building industrial applications and products using ROS2 [23].

3.1.2 Navigation2

The ROS Navigation2 Stack (Nav2) combines different packages that allow mobile robots to navigate from point A to point B. Navigation2 is the de-facto standard for mobile navigation with a wide range of supported robots. Supported robot types are holonomic, differential-drive, legged, and ackerman (car-like). For the mobile robot to make use of the Nav2 stack, it has to be set up in a certain way to be able to generate plans and execute commands in the right way. Nav2 requires the mobile robot to possess a laser scan or point cloud sensor, an odometry source (such as an IMU or wheel encoders), and a set of transformations for planning and navigation. The stack contains tools that save and load occupancy maps, localize the robot, plan paths, execute the path, provide cost maps, build behaviors and execute recoveries [24].

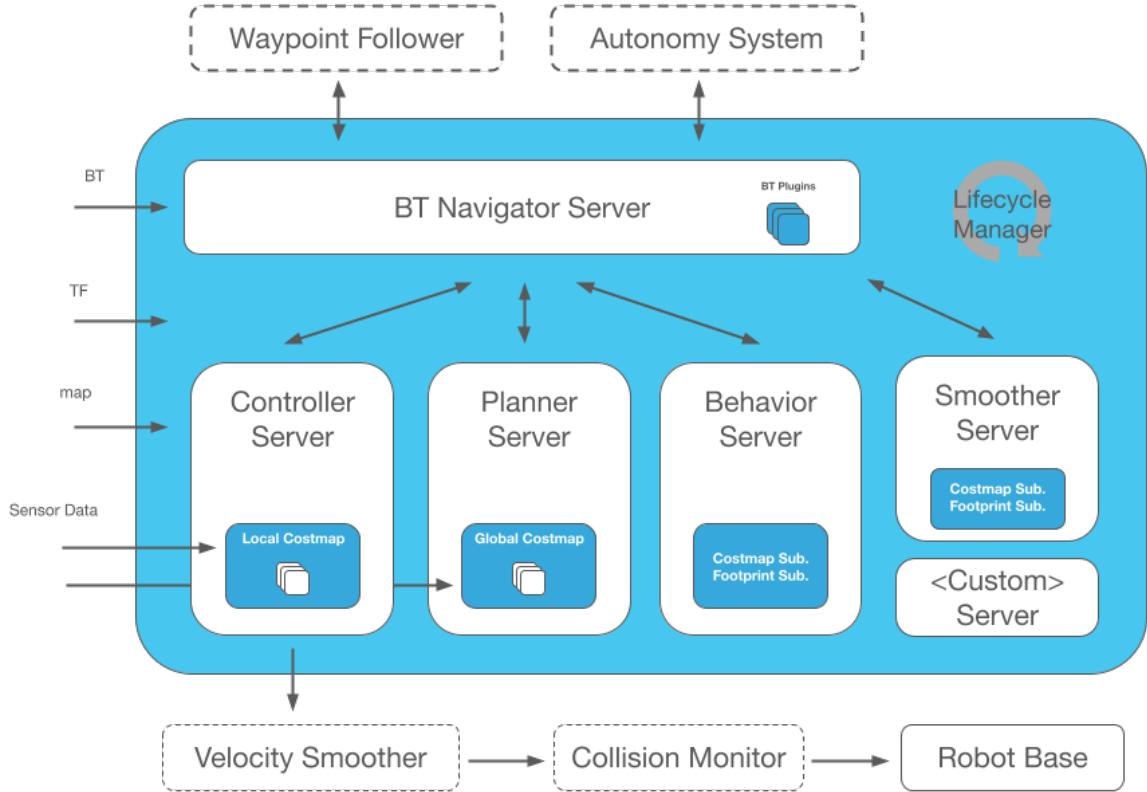


Figure 3.1: Navigation2 Architecture [24]

These packages are often integrating a SLAM method to build or enlarge maps. Path planning and execution capabilities, which correspond to the global and local planner described in section 2.2, are further supported by ready-to-use planning plugins that use A* and Dijkstra algorithms for path planning and a dynamic window approach for path execution. The system sequencing is done in a hierarchical structure as a central behavior tree calls asynchronous actions from the respective planners after another, as seen in figure 3.1.

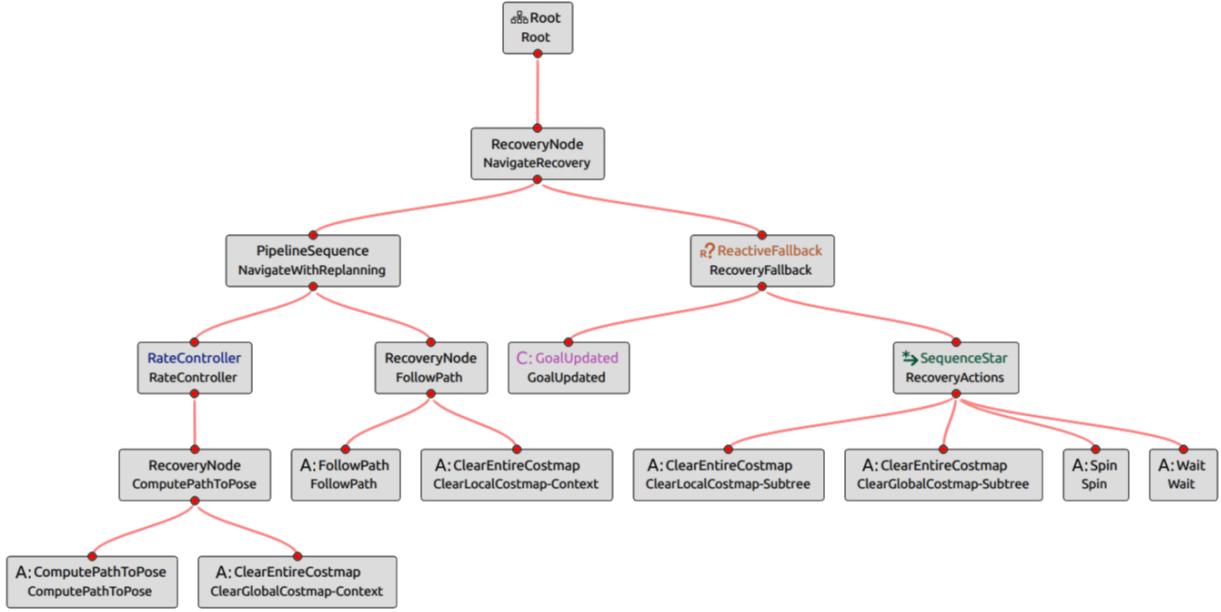


Figure 3.2: Navigation2 Behavior Tree

The behavior tree depicted in figure 3.2 is the one that is the default Nav2 behavior and has a set of recovery behaviors already implemented (spin, wait, back up, clear map). These behaviors are executed if the robot can not make progress towards the set goal. Nav2 uses a node lifecycle management system to start, activate, deactivate, and shut down nodes in a controlled way. The lifecycle approach allows the system degree to monitor the system node's startups, executions, and failures.

With this quantity of functions and hierarchical architecture, the ROS Navigation2 stack is an excellent basis for achieving high levels of robot autonomy. The software can be modified and expanded with plugins to fit the users' needs. When comparing the functionalities of the software stack to the proposed autonomous system architecture in figure 2.2, the things that Navigation2 is missing are a system-wide supervision layer and a more deliberative approach to behavior planning, as mentioned in chapter 2.3.

The absence of these components leads to a limited set of circumstances in which the robot can perform navigation reliably. One of the core problems behavior-based robots face is generating a dependable environment representation. An accurate representation is the basis

for the last steps in the planning part of the "Sense - Think - Act" cycle. Unforeseen events can severely limit the reliability of the environment representation without the robot being aware of it. Examples of such events currently not handled appropriately are slipping wheels, orientation changes by external influences, or undetected, more specifically, undetectable obstacles. This can lead to erratic movement of the robot, as reactive behaviors and more deliberative behaviors compete for authority over the robot's movement. By design, to allow more intelligent robot behaviors, the deliberative behavior can override commands from reactive behaviors. However, the deliberative behavior's planning is based upon a false representation of reality, leading to unsafe commands. The unsafe planning triggers reactive-type behaviors again to counteract the commands from the deliberative behavior. Even worse would be that the reactive behaviors may not get activated, and the robot continues to drive despite being unaware of the surroundings. The unsafe behavior requires a human operator would need to step in and restore the robot to full functionality by hand.

3.1.3 Turtlebot3

The Turtlebot3 is a standard mobile research platform with an available ROS interface to control the robot. The robot is well integrated into the ROS ecosystem and is established in the literature as a system to develop and integrate new methods. The robot has two motors with attached wheel encoders to drive and steer, classifying it as a differential drive robot. A third omnidirectional wheel stabilizes the robot.

The manufacturer provides an open-source model for simulating the robot in physics-based simulators like Gazebo, as depicted in figure 3.3, which enables faster development and testing for the robot.

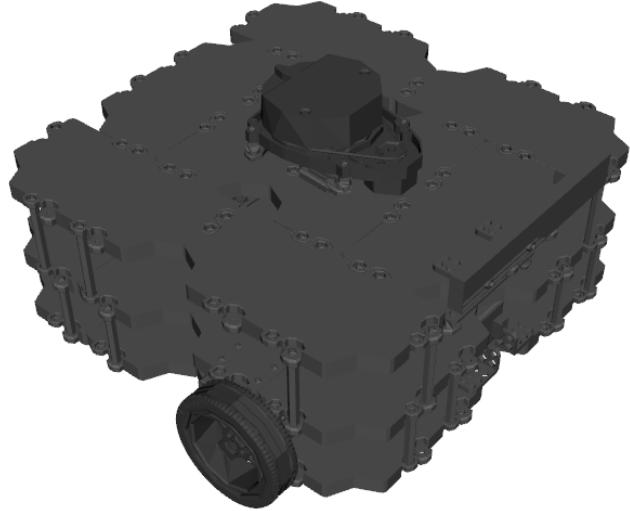


Figure 3.3: Turtlebot3 model in Gazebo simulator

The equipped sensors and easy simulation capabilities make the robot well suited for using the Nav2 stack and further development for behavior planning.

The robot specifications are specified in table 3.2.

Table 3.2: Turtlebot3 Specifications

Specification	Value
Maximum translational velocity	0.26 m/s
Maximum rotational velocity	1.82 rad/s (104.27 deg/s)
Maximum payload	30kg
Size (L x W x H)	281mm x 306mm x 141mm
Weight	1.8 kg
Singe Board Computer	Raspberry Pi (3 or 4)
Laser Distance Sensor	360 Degree Laser Distance Sensor LDS-01
IMU	Gyroscope 3 Axis, Accelerometer 3 Axis
Actuators	XM430-W210

The turtlebot will be used for the analysis of the current state of mobile navigation. Furthermore, the simulated turtlebot will be the platform on which the behavior planning will be implemented and tested.

3.2 Requirements

The software requirements that improve the robot's behavior are derived from the current limitations of the standard ROS and Navigation2 setup for mobile robots described in the previous chapter. The functional requirements and their acceptance criteria are listed in 3.3. The non-functional requirements are listed seperately in 3.4. To better contextualize the context of the systems requirements, figure 3.4 depicts a Use-Case diagram with the needed functions to improve the system's autonomy.

3.3 System Solution

For a Turtlebot3 running Nav2 to appropriately deal with the listed requirements, the system needs to be extended with new external modules. To eliminate a single point of failure for the system, one needs to create an independent second system outside the existing one. This is to create a robust fallback behavior that does not rely on Nav2 to move. Also, an external system can more reliably monitor other systems when its execution is detached from the performance and execution of other systems. A suitable name for the second system is the "autonomy layer", as it aims to increase the robustness and autonomy of the whole system. The autonomy layer has to provide a comprehensive system supervisor

Table 3.3: Functional Requirements

Nr.	Name	Priority	Description/Acceptance Criteria
fn_req1	Sensor Failure	High	The system detects sensor failure. Ensure that the system can restart sensors and decrease the speed during the time the sensor delivers limited information
fn_req2	Emergency Detection	High	The system detects emergency. Ensure that the system can detect when the continuation on the calculated path is no longer safe (sensor failures, blockage)
fn_req3	Emergency Stop	High	The system can initiate emergency stops. Ensure that the system can override all commands and stop in case an emergency is detected
fn_req4	Override Navigation2	High	The system can override navigation2. Ensure that the system's commands can always override the commands coming from navigation2.
fn_req5	Maintain operability	High	The robot executes commands as long as it is safe. Ensure that the robot keeps driving if it is safe even when system functions are not working correctly.
fn_req6	Recovery	High	The system can recover from crashes. Ensure that the system can successfully reach goals despite previous crashes
fn_req7	Control Path Planning	Medium	The system controls and rates the quality of the planned paths.
fn_req8	Reset Goals	Medium	The system can reset and override goals set by the user so that the goal is reachable by planners.
fn_req9	Robot Range	Medium	Ensure that the robot will not run out of battery during navigation to a goal.

that monitors all components' health (sensors, navigation, robot controller). The constant execution monitoring equips the system with the ability to deal with the event of node failures and component malfunctions.

Another required addition to the autonomy layer is a data storage of relevant system information. The stored data includes sensor data and generated maps, cost maps, positions, and speed commands. Using past data points allows behaviors to become more deliberative in their approach and opens up many possibilities for intelligent behaviors like movement predictions of obstacles.

A third component to the autonomy layer is a behavior planner largely independent of Nav2 execution and planning. This behavior planner must be capable of overriding the Nav2 behaviors if needed. The component processes information from the execution checker, Nav2, and past and present sensor data to decide if and how to override the default behaviors. Based on this information, the behavior planner can make a deliberative, strategic decision

Table 3.4: Non-Functional Requirements

Nr.	Name	Priority	Description
non_freq1	Single Point of Failure	High	The system does not have a single point of failure When parts of the system fail, the system maintains operability to a degree
non_freq2	Performance	High	The system's control loop guarantees fast reactions. The average frequency with which the system checks the sensors/navigation2 is higher than 100Hz (10ms)
non_freq3	Determinism	High	The outcome for a given set of inputs must be deterministic, meaning that the behavior is always executed similarly.
non_freq4	Deliberate	High	The robot can execute deliberative behaviors, meaning that behaviors new implemented behaviors go beyond reacting to sensor input and have a planning aspect.

to increase robustness and autonomy, thus decreasing reliance on human supervision.

Additionally, a component to switch intelligently between speed commands from Nav2 and the autonomy layer is needed. This component could be realized as a decision gate that receives speed commands from Nav2 and the new behavior planning component and can forward, block or modify commands sent to the controller. This modification aspect allows high-quality local planning capabilities but enables more deliberative behaviors to happen upon this planning. A simplified system block diagram is depicted in figure 3.5. The proposed additional components, discussed in this chapter are colored in blue. Existing components, discussed in chapter 3.1, are colored in gray.

3.4 Software and Simulation

A comparison and assessment of the behavior planning approach were made in chapter 2.4. The decision on which approach to take fell on the behavior tree method. The reason for this is mainly the increased flexibility when creating and expanding the behavior planning compared to FSMs. Also, behavior trees allow complete determinism and introspection which is why POMPDs are not suitable as the behavior planning approach for the specified requirements.

3.4.1 Behaviortree.CPP

The Navigation2 stack uses a behavior tree to coordinate the planning layer. The behavior tree that is used is an expansion of the Behaviortree.CPP (BT.CPP) library. This library

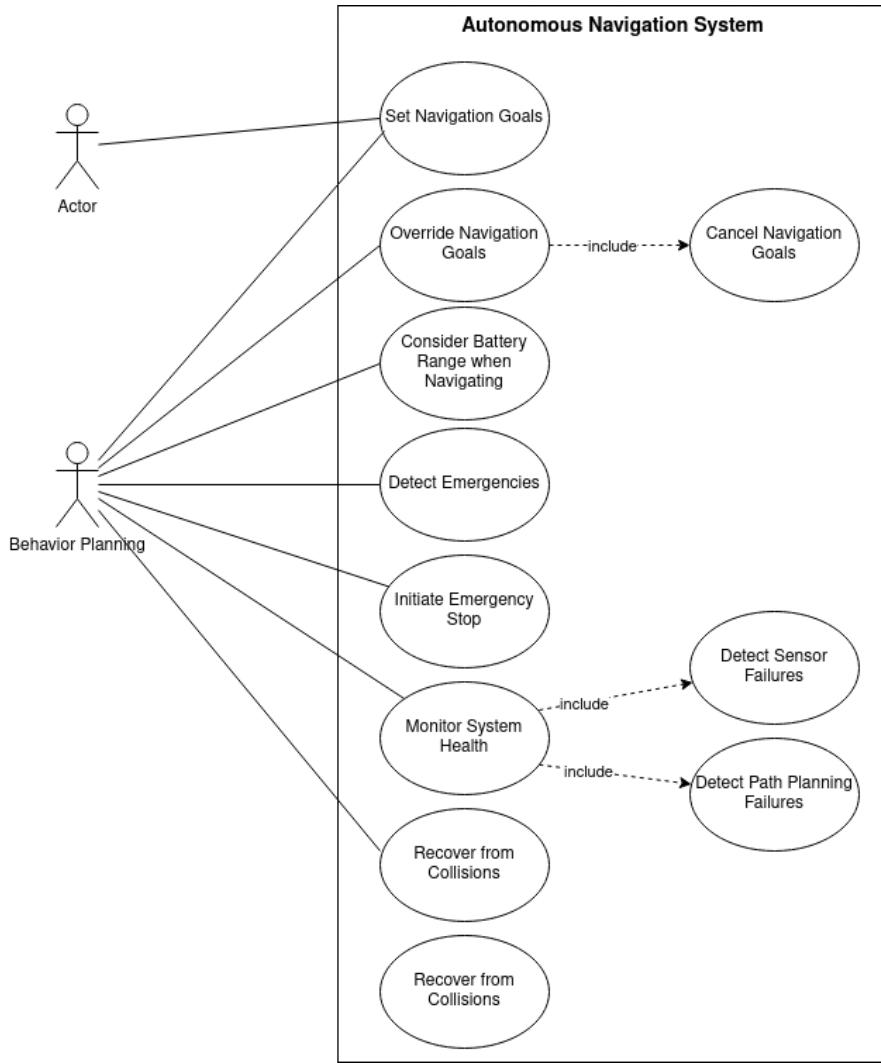


Figure 3.4: Use Case Diagram

offers a fast way to create, execute, monitor, and edit behavior trees. Additionally to the types of control nodes that are mentioned in chapter 2.4.2, the library adds the concept of reactivity into the catalog of control nodes. Reactive sequences and reactive fallbacks differ from normal ones in handling nodes that return the state "running". Instead of ticking the node again, the whole sequence restarts, which is very useful for continuously checking a condition and executing an asynchronous action node that gets halted when a condition is not met anymore (compare table 3.5).

Also, it offers Decorator Nodes which allow more control over the child node and its output. These Decorator Nodes can only have one child node each. A comprehensive list of the available Decorators and descriptions is presented in table 3.6.

To allow the tree nodes to communicate, the library provides the developer with two possibilities. Either node can use the blackboard, a dictionary (key/value) that all nodes of

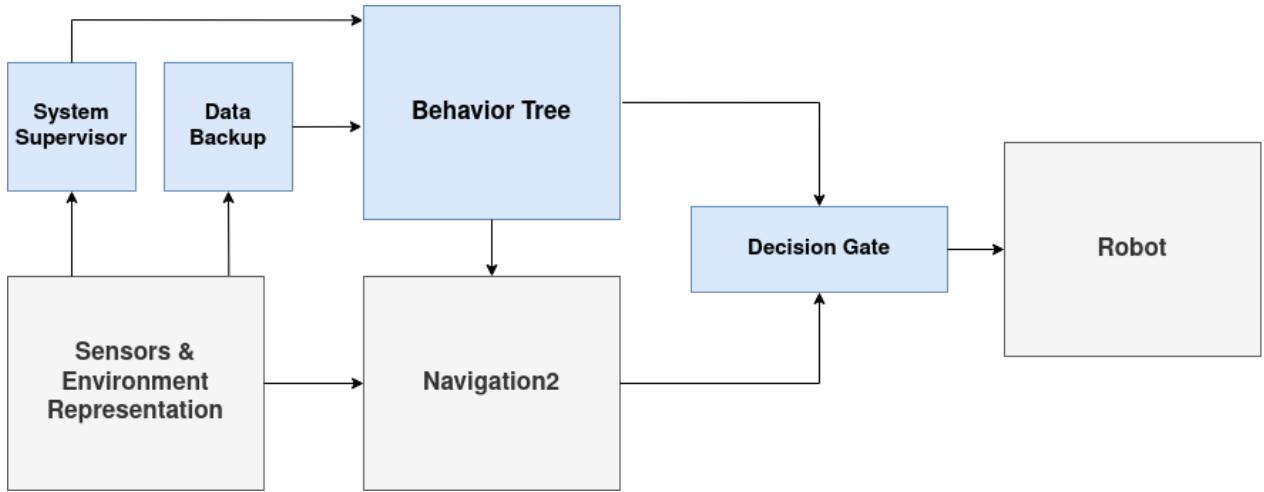


Figure 3.5: System Block Diagram

Table 3.5: Control Nodes in BT.CPP

Name	Child Returns Failure	Child Returns Running
Sequence	Restart	Tick Again
Reactive Sequence	Restart	Restart
Sequence Star	Tick again	Tick again
Fallback	Tick next	Tick again
Reactive Fallback	Tick next	Restart

the tree can read and write. Alternatively, two nodes can be connected through ports which allows direct communication between two nodes via a key/value.

3.4.2 Groot

Groot is a software to create, edit, monitor, and debug behavior trees with a graphical user interface. The software allows the creation of behavior trees in XML files which can be directly loaded into and used by the BT.CPP library. Groot can monitor the live execution of behavior trees and allows the introspection of how the tree reacts to different scenarios (compare fig. 3.6).

3.4.3 Gazebo Simulation Environment

The development and testing of behaviors will be done with the Gazebo simulator and the Turtlebot3 model. Gazebo is well-integrated into ROS and ROS2 and offers many ROS interfaces to control the simulation. Gazebo accurately models the physics of the robot and can simulate and publish the wheel odometry directly on a ROS topic. Furthermore, the simulator can simulate the readings for the laser scanner and IMU data via plugins. The

Table 3.6: Decorator Nodes in BT.CPP

Name	Succeeds	Fails	Running
Inverter Node	If the child returns false	If child return success	If the child returns running
ForceSuccess Node	Always	Never	If child returns running
ForceFailure Node	Never	Always	If child returns running
Repeat Node	Ticks child as long as it returns success. The maximum number of repeats can be defined	If the child returns failure	If the child returns running
Retry Node	If the child returns success	Ticks child as long as it returns failure. The maximum number of ticks can be defined	If the child returns running.

use of the simulator allows easier repeatability for test cases as various obstacles can be spawned at any time, and sensor failures can be easily induced, resulting in quicker overall development. Using a simulator allows slowing down or speeding up the environment time. The time control allows more intricate observation of fast reactive type behaviors in slow motion while also enabling the observation of the long-time robustness of the system during extended tests. The ROS community offers many different simulation environments to test the robot in. The environments range from simple geometric structures, to office spaces up to complete race tracks.

3.4.4 OS and Software Versions

The software versions used to implement and test the system on are listed below in table 3.7. Many of the software versions are determined by the choice of the ROS distribution (distro). ROS2 Foxy is not the newest ROS2 distribution but is well supported and developed, so the decision was made to use this distribution. The decision why the turtlebot is used is laid out in chapter 3.1.3. Although Python and C++ can be used to develop applications with ROS2, the behavior tree library relies on C++, which is why C++ was used for developing the whole system.

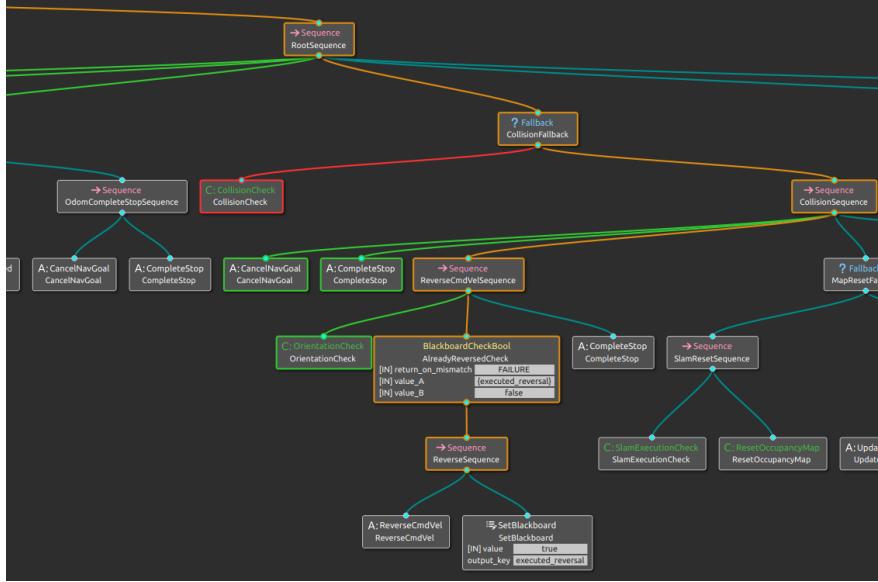


Figure 3.6: Live monitoring of behavior trees with Groot

Table 3.7: Used Software Versions

Name	Version / Release	Comments
Operating System	Ubuntu 20.04 LTS	
ROS	ROS2 Foxy	Long-Term-Supported Distribution
Navigation2	Foxy-devel	Newest Release for ROS2 Foxy
Simulation	Gazebo 11	Determined by choice of ROS Distro
Robot	Turtlebot3	
Behavior Tree	BT.CPP 3.7	
Programming Language	C++ 14	Determined by choice of ROS Distro
Build System	CMake and Colcon	Determined by choice of ROS Distro
Development Environment	Visual Studio Code	Offers IDE debugging options for ROS

Chapter 4

Implementation

4.1 System Architecture

The system architecture, depicted in figure 4.1, integrates the behavior tree into the existing architecture. The already existing components are colored in gray, and the newly added components are colored in blue. The new components and their functions are described in the following sections of this chapter.

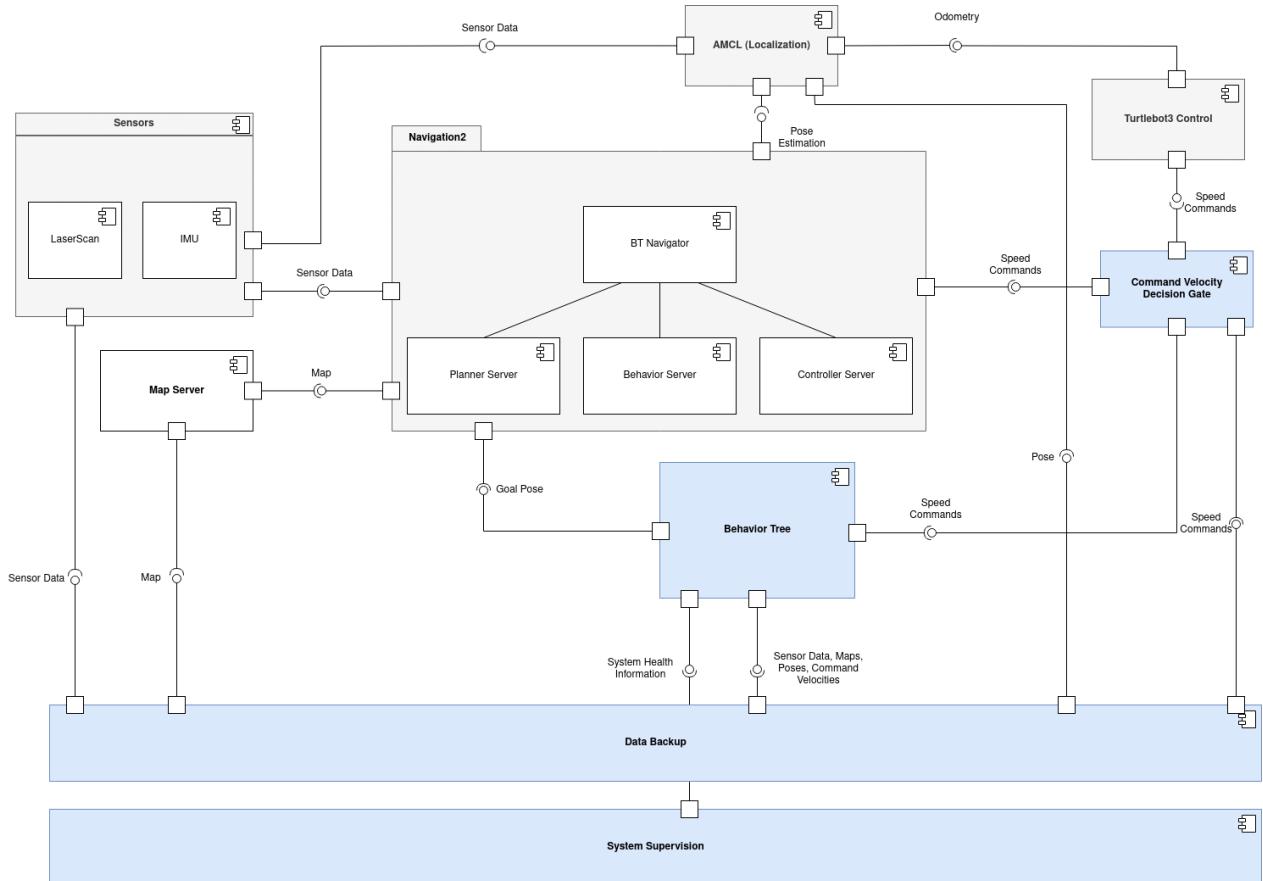


Figure 4.1: Component Diagram of System Architecture

4.2 Data Backup

The data backup component functions as a way to efficiently gather data from all system components and provide necessary data to the behavior tree when the planning process requires past data points. This component subscribes to all sensor data and saves the messages in a queue data structure. The most recent data gets pushed into the array, and if the array exceeds a specified length, the oldest data this data gets deleted from the array. There are two main reasons to limit the array size in this way. On the one side, the goal is to decrease the size of the message that gets sent to the BT to limit the network load, and on the other side, the BT often does not need to look back further than a few seconds to make a decision. This way, the computational load, and the network load are decreased.

The BT can access the saved data by sending a service call to one of the provided services by this component. Service calls in ROS2 are executed with high priority and are done quickly. However, it adds steps to the algorithm instead of just saving the data directly in the behavior tree. The reason for outsourcing the data backup to another component is that the ROS node needs an executor which constantly spins the node. The spinning action checks the network to see if there is any work for the node. Work, in this context, means that if one of the subscribed topics receives a message, the node has to execute one of its callback functions associated with that subscription. This spinning has to be done in regular intervals. Otherwise, the node might miss incoming messages from subscribed topics. The way to do the spinning is to have the node constantly spun in one thread. Having a constant spinning node would completely block the behavior tree's execution. If one tries to spin a node once when a node gets ticked in the BT, the problem of missing messages will become relevant because the BT might need longer than expected to return to a node to tick it. That way, the BT node responsible for saving all the relevant data can not guarantee that it has picked up on all the messages between spins. The synchronous nature of the ROS executor spinning the nodes is why the data backup is not stored within the BT. The saved data inside this component with the corresponding lookback time is listed in table 4.1.

Table 4.1: Types of saved data

Name	Type	Lookback Time
Lidar	Laserscan	3 seconds
Poses	Pose with Covariance	2 seconds
Map	Occupancy Grid	Only saved last one
Collision Pose	Pose with Covariance	Only last one
Command Velocities	Twist	2 seconds
Global Costmap	Occupancy Grid	Only last one

4.3 Command Velocity Decision Gate

This component is responsible for modifying or blocking the Velocity Commands from Nav2. The component receives messages from Navigation2 on the ”/cmd_vel_nav” topic and the ”/cmd_vel_bt” topic from the behavior tree. This component publishes the received messages on the actual topic ”/cmd_vel” which is subscribed by the robot controller. The BT can modify the Decision Gate parameters to either disregard incoming messages of Nav2 entirely or to modify them. The modification is helpful when the situation demands the robot to slow down. If the Decision Gate receives messages from Nav2 and the BT, it will prioritize the BT. This situation should theoretically never occur as the BT will set the parameters for the Decision Gate accordingly and cancel the current Navigation Goal before publishing commands on its own. It is implemented as an additional safety measure in case the cancellation of the goal is not possible if Navigation2 is not responsive to the cancellation command.

4.4 Behavior Tree Structure

A simplified behavior tree only depicting the top-level behavior control fallback nodes is shown in figure 4.2. The tree receives the initial tick signal from the root in an infinite loop.

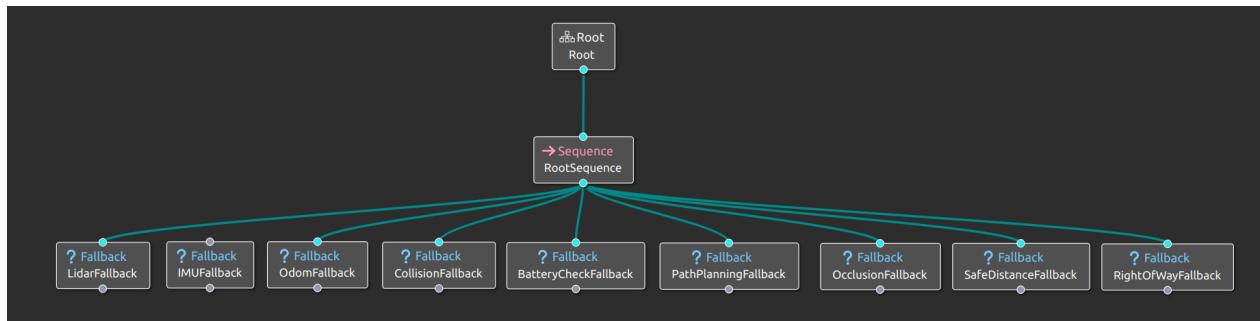


Figure 4.2: Top Level Behavior Tree Structure

The root control node is a sequence node. The root node is a simple sequence because the whole system must function with increasing complexity in the tree’s later parts (on the right side). The complex behaviors require that all previous components and conditions work as they should. The root sequence will not allow the execution of higher-level behaviors if the system detects problems on more fundamental levels. The root sequence is the first step for increasing the system’s robustness to combat competing reactive and deliberative behaviors that could endanger the robot’s safety and other actors in its environment.

4.4.1 BT System Supervision

The design pattern of the system supervisor is an ”Watchdog” pattern. This pattern is depicted in figure 4.3. The pattern is used to monitor a system from the outside and directly take corrective measures when problems are detected.

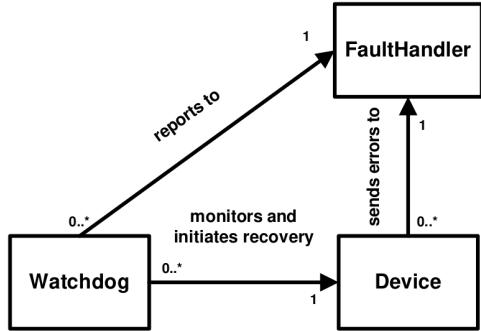


Figure 4.3: Examiner pattern for the system supervisor component [25]

The system supervisor component is partly outsourced into a component outside of the BT's thread for the same reasons as the sensor backup component discussed in the previous section. The synchronous executor would stop the execution of the whole behavior tree, which is why the component needs to be moved inside of a new thread. The system supervisor, also called the execution checker, is receiving messages from nodes not implemented as ROS2 lifecycle nodes. If a node stops sending messages or responding completely, the external execution checker can inform the BT via a provided service of a problem with the node. ROS2 lifecycle nodes provide information about their health via an FSM implementation inside the node. The current node state can be requested via a service call. However, the node's output needs to be checked, too, because it might be that the FSM inside of the node did not pick up on an error and failed to change the internal lifecycle state. The system supervision is constantly checking the system's health. The BT is using service calls to get the health information about the components checked inside Condition Nodes. The components get checked by the BT inside of a Fallback. If the condition for the execution is met, the Condition Node returns "Success", and the fallback would exit to move on to the next fallback and execution condition check (see fig. ??). If the execution checker detects a node failure, a sequence to react to the sensor failure is executed in which the robot slows down to a minimum and tries to restart the sensor. Should the restart fail twice, the robot will not be able to navigate safely anymore and will come to a complete stop. This behavior pattern is realized for the lidar, the IMU, and the wheel odometry.

The timeout period for the execution checker to declare a node failed after receiving no message is set at one second. After that period, the goal is to enable the robot to continue progressing towards the goal. However, if the system can not restart the sensor in a short time after failure, the robot has to come to a stop as a sensor failure is a severe function loss for the robot.

4.4.2 BT Advanced Behaviors

After the BT has checked for sensor failures and can guarantee that the environment representation is accurate, the system can check for more complicated scenarios to improve autonomy. The implemented behaviors are outlined in this section.

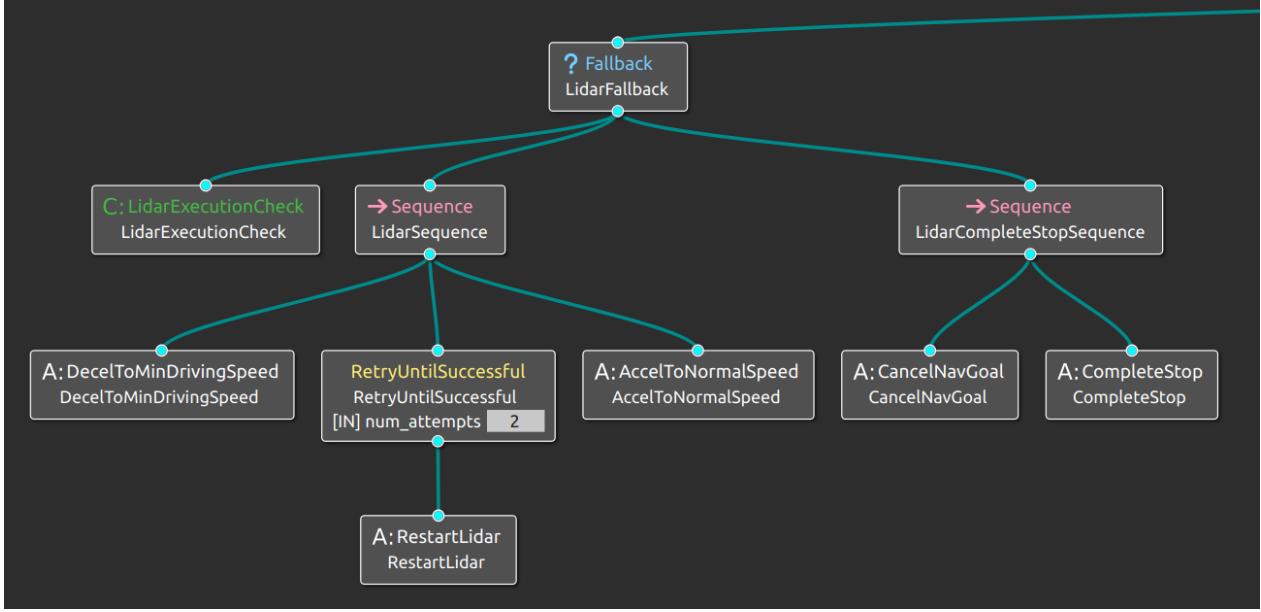


Figure 4.4: Lidar Fallback

Collision Behavior

One of the biggest problems mentioned in chapter 3.1 is the inability to react to collisions. The collision can happen due to inadequate sensory coverage for detecting obstacles or general sensor inadequacy, which can not detect obstacles because of their shape or surface material. Collisions would be improbable with many sensors covering all angles and heights and computer vision capabilities to detect obstacles. However, even if the robot could detect all obstacles, it would still be vulnerable to outside perturbations, e.g., other robots driving into it or humans accidentally touching the robot. In order to combat this weakness, the robot must be able to detect that a collision occurred, get out of the collision state, update the map and reset all costmaps to generate a new path to the original goal.

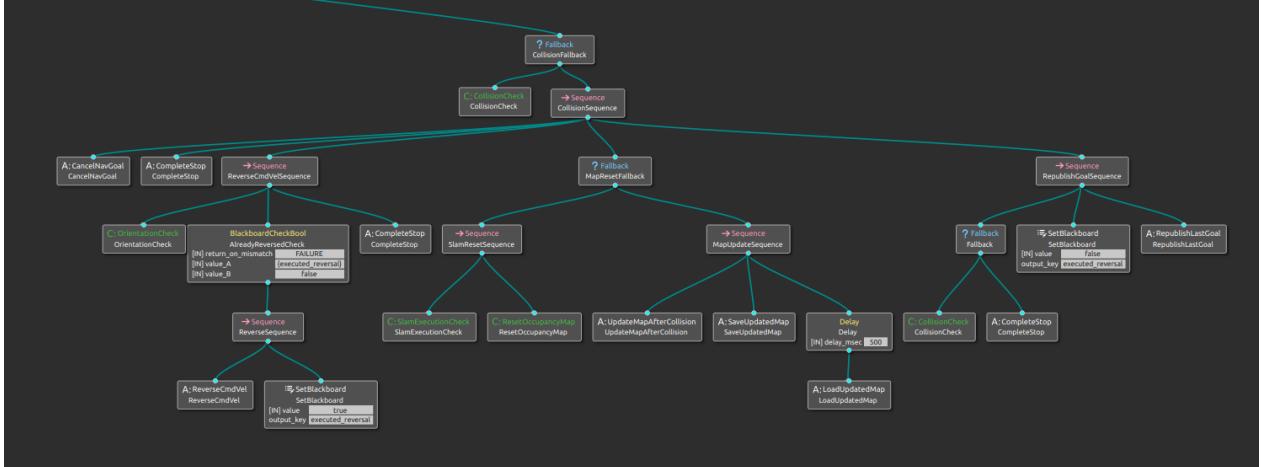


Figure 4.5: Collision Fallback Behavior

The collision checker uses a simulated sensor to detect collisions. On a real robot, the collision checker can use an IMU-based collision detection, but this has not been implemented in this behavior tree due to the time constraints of this thesis. The collision checker is integrated into the system supervisor, and the BT makes service calls to the execution checker to get the collision state of the robot periodically. The child nodes of the collision sequence use the sensor data backup. To get out of the collision state, the BT requests the last saved command velocities from the data backup component and modifies them to reverse with minimum speed the same way the robot got into the collision. The reversing action takes longer than the two seconds as the commands get published with more extended time intervals between them to adjust the driven distance to be the same as the navigation commands. An additional check is implemented before the reversing to ensure the robot only reverses once. Otherwise, the robot would drive back and forth as the data backup service would provide the BT with the commands from the last reversing action.

Regarding the map reset, it has to be determined if a new map is being created with a SLAM method or if the map server provided a previously saved map. In the case of the running SLAM, the recent updates to the map must be discarded, and a previous map version before the collision has to be re-utilized. When the map server component from Nav2 provides a static map, the map needs to be updated with a new obstacle at the point of collision and republished by the map server. The activity of the "MapUpdateAfterCollision" action node is depicted in figure 4.6. The collision point is estimated by the robot's pose, which refers to the robot base link and the robot footprint. It is assumed that the collision point was right in front of the robot because the local planner favors forward drive kinematics and probably is driving forwards when a collision occurs. The simulated bumper sensor can provide more nuanced information about the point of contact, but this functionality can not be achieved on a real robot with IMU-based collision detection. Hence the obstacle position is simplified to always be in front of the robot.

A 25 by 25 cm square with 100 percent occupation at the collision point is added to the current map and then saved to be reloaded by the map server.

The original goal is then republished to replan the global path on the updated map. A more detailed view of the whole behavior is depicted in figure 4.6.

This way, obstacles lower than the lidar scan height can be detected. With this behavior, the robot should be able to eventually navigate around transparent obstacles that do not reflect the laser. Glass doors are known to cause issues for robots based only on lidars.

Battery Behavior

The battery behavior has the goal of monitoring the robot's battery and has to decide if the robot should navigate to a new goal location or not. The decision is based on the current battery charge and the estimated consumption of energy to reach the new goal. The robot should not begin the navigation process if it is probable that the remaining charge will not suffice for the driving. Figure 4.7 depicts the behavior tree fallback.

The robot has to consider the idle consumption and the energy to drive the motors. With this information, a linear function for energy expenditure was created to predict battery usage depending on the path length. An additional safety factor was added to the energy estimation function to consider the scenario in which the robot must reroute and drive a longer route or stop for a moment. The safety factor was included to increase the behavior's robustness and cover more scenarios.

The average values for idle and driving consumption on a real robot could be experimentally determined. For the simulated turtlebot, a simulated battery package was created, which updates the battery charge every second. The battery package subscribes to the command velocity topic and decreases the battery charge for every message received. The battery package was created because the behavior for this scenario needs to be tested, and Gazebo does not offer this function. The consumption rates are not close to reality but are chosen to allow the rapid testing of the behavior. The battery package provides a service to get the battery charge and the values for the assumed consumption rates while idle and driving (compare 4.2. The BT node "IsBatterySufficientCheck" calls the service to compare the current battery to the calculated consumption. The battery behavior fallback will cancel the navigation goal if the robot can not reach the given goal with the remaining charge.

Path Planning Behavior

With this behavior, a way to make the global planner more flexible was implemented. The execution checker component monitors the health of the Nav2 global planner. Additionally, the behavior tree analyses the global planner's output. If the latter condition fails, the given navigation goal is not reachable by the planning algorithm. By shifting the goal to a nearby alternative goal, the robot tries to navigate, although the original goal is not reachable. If the location is still unreachable, the goal gets moved further away from the original. The routine repeats up to ten times, leading to a maximum distance to the original goal of up to one meter. The goal gets moved on a straight vector towards the robot. Other strategies for finding alternative goals were not implemented in this behavior. Figure 4.8 depicts the behavior tree fallback for the behavior.

4.5 System Support Tools

4.5.1 Gazebo Sensor Drivers

For testing purposes, the simulated sensor data is not published directly on the actual topic by Gazebo. An intermediary node is used, which subscribes to the Gazebo topic and republishes the information on the original topic. The failure of a sensor can be simulated this way by stopping the execution of the intermediary node. Figure 4.9 shows the RQT graph, which depicts the ROS nodes and their subscribed/published topics.

4.5.2 Custom ROS Interfaces

Because of the necessity to have ROS nodes spinning outside of the BT thread, as discussed in 4.4.1, additional interfaces needed to be created in order to receive information through service calls. The created service types are listed in table 4.2. All service definitions are created in a dedicated package named "bt_msgs".

Table 4.2: Implemented Custom Services

Name	Request	Response
GetCharge	Empty	float charge, idle_decrease_per_sec, drive_decrease_per_sec
GetDistance	Empty	float distance_in_meter
GetLastGoal	Empty	geometry_msgs/Pose last_goal_pose
GetLastMap	Empty	nav_msgs/OccupancyGrid map
GetTwistArray	Empty	geometry_msgs/Twist[] cmd_vel_array
PubCmdVel	geometry_msgs/Twist cmd_vel, float time_in_seconds	bool success
SendUpdatedMap	nav_msgs/OccupancyGrid updated_map	Empty

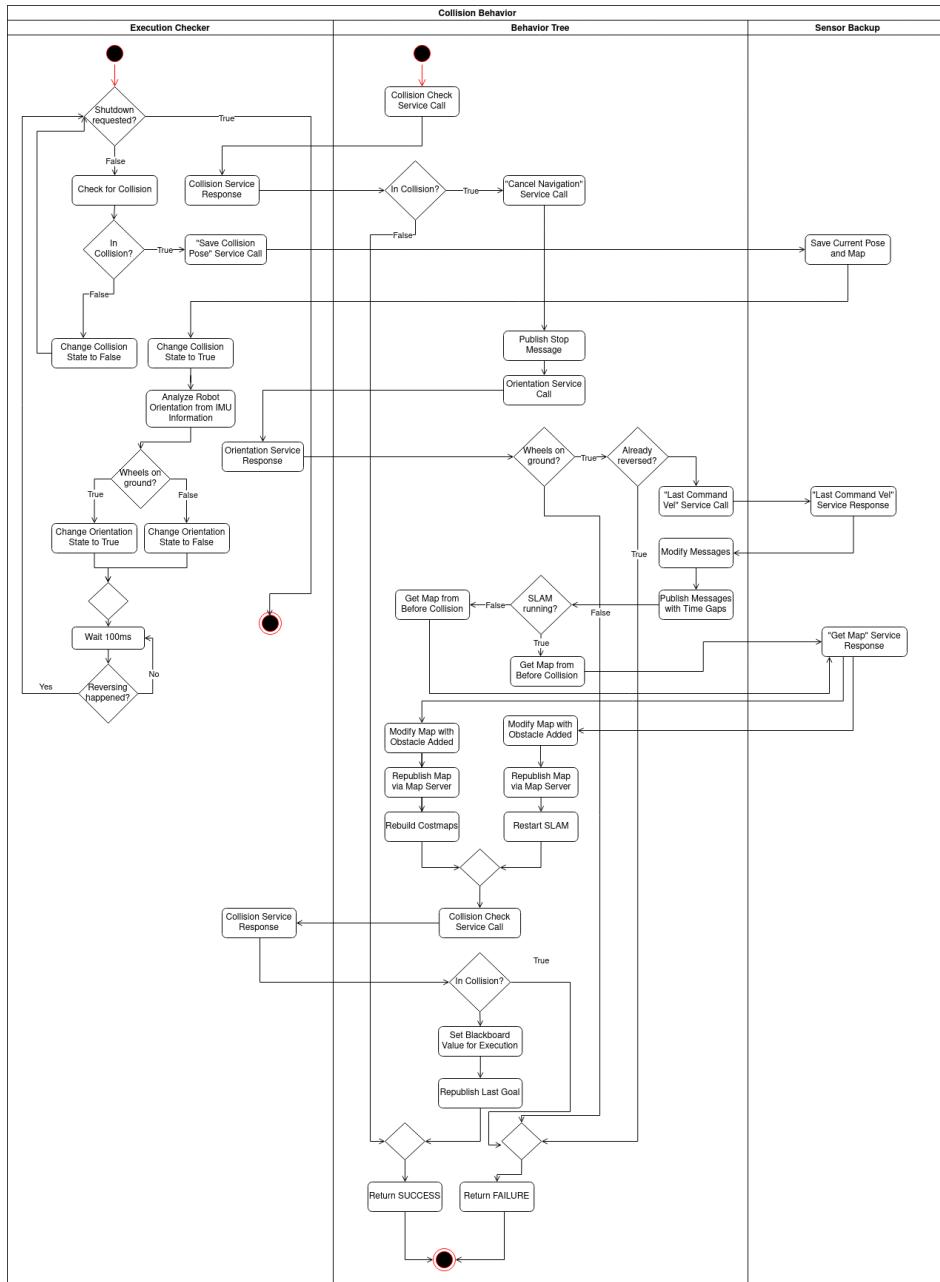


Figure 4.6: Activity Diagram for the Collision Behavior

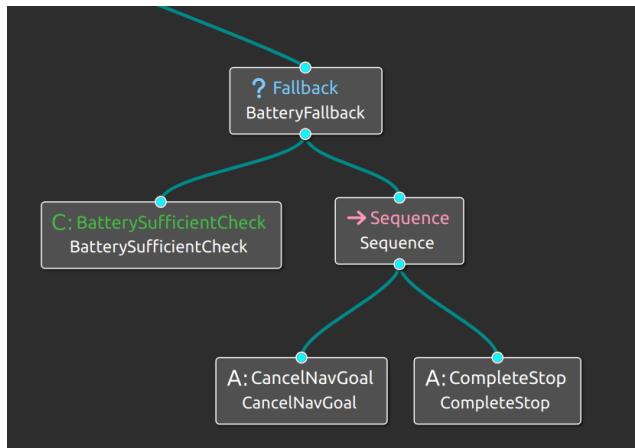


Figure 4.7: Battery Fallback Behavior

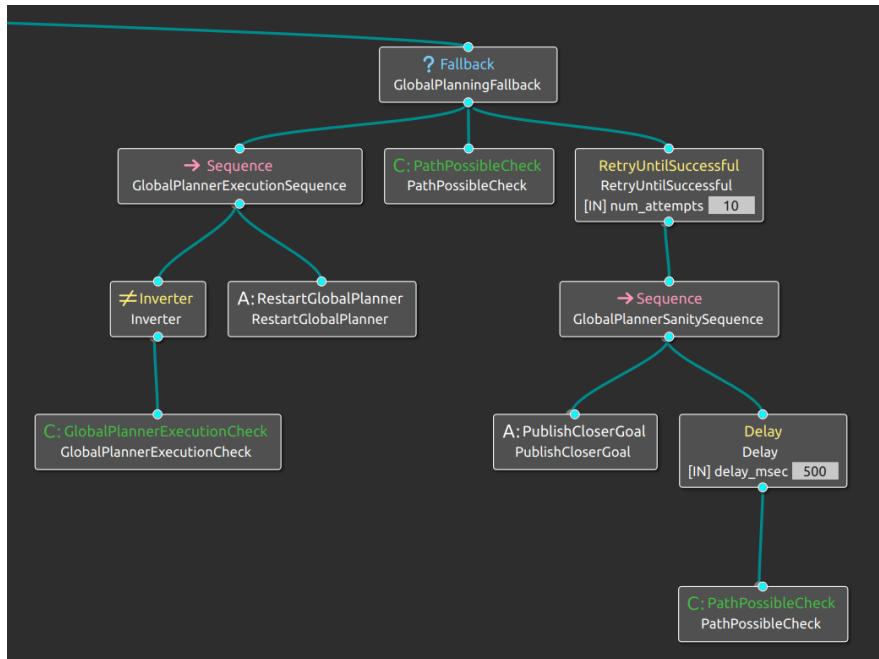


Figure 4.8: Path Planning Fallback Behavior

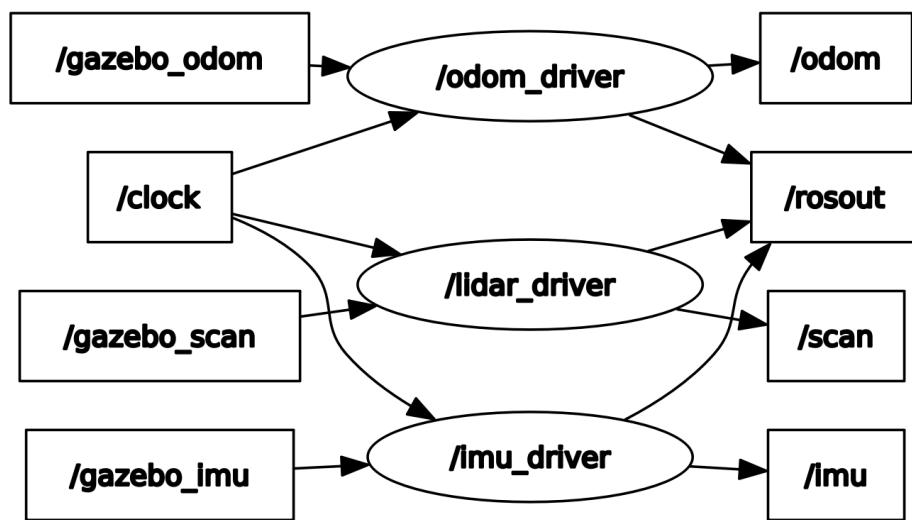


Figure 4.9: Graph for the Sensor Driver nodes and topics

Chapter 5

Evaluation

The robot is put into specifically designed scenarios that should trigger behavior responses from the behavior tree component. The scenario testing is done to evaluate the efficacy of the implemented behaviors in relation to the functional requirements listed in chapter 3.2.

The scenarios are executed in a simulated apartment environment as seen in figure 5.1 and a simulated world with small round obstacles in an enclosed space as depicted in figure 5.2.



Figure 5.1: Apartment Environment in Gazebo

The environments were mapped beforehand with the ROS2 "Slam Toolbox" package and provided via the Nav2 map server.

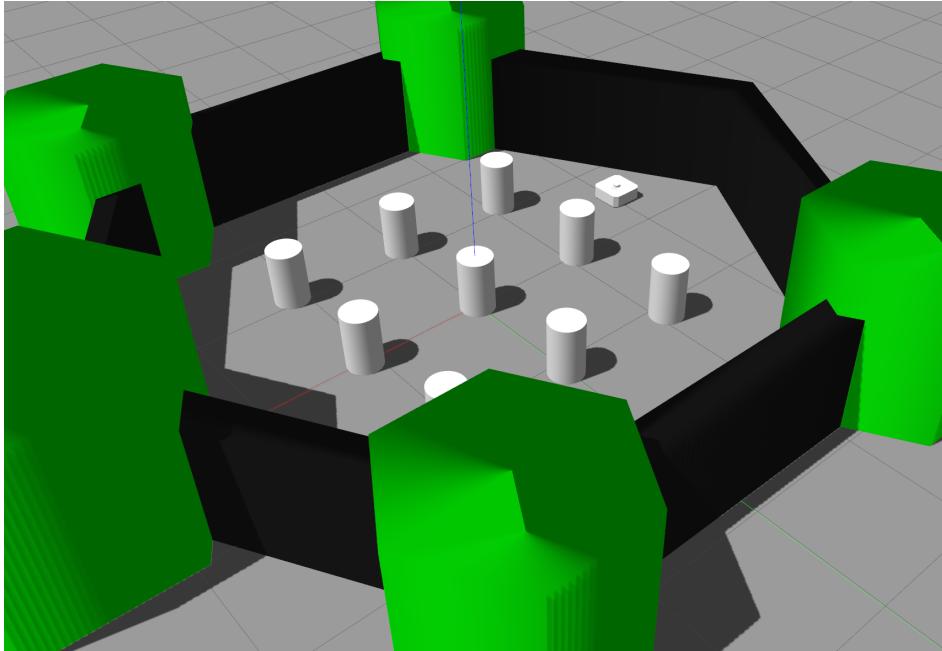


Figure 5.2: Enclosed Environment in Gazebo

The robot location and navigation goals are defined in another ROS Node, which programmatically spawns the robot into the environment and sets the goal this way. This way, the same scenario can be run with and without behavior planning. To test the behavior of the sensor failure fallbacks, the respective sensor driver, mentioned in chapter 4.5.1, was shut down to emulate the sensor's failure.

5.1 Scenarios

The scenarios are derived from the functional requirements and can be used to test multiple requirements as listed in the tables 5.1 and 5.2. The acceptance criteria often contain multiple binary conditions that can only be met or failed. All acceptance criteria for a given scenario must be met for the test to be counted as a success.

For the Lidar, IMU, and Odometry scenarios, the sensor driver was shut down when reaching the desired speed listed in the scenario. Scenario NoPathFound1 is a scenario in which the lifecycle state gets transitioned into "inactive" and can not execute the planning action anymore when triggered. The BT has to recover from the inability to plan in this scenario. NoPathFound_2 is simulating the scenario when the planner is active but fails to find a collision-free path to the desired goal. This scenario tests the ability to find alternative goals and navigate to a nearby goal. In the Collision_1 scenario, an obstacle gets moved so close to the robot that a collision is registered. The forced collision tests the ability to escape a collision and resume navigation afterward. A new navigation goal is published after the behavior tree exits the Collision Fallback Routine. Collision_2 scenario is testing the same behavior but during active navigation. A flat obstacle is spawned in the path of the robot. The obstacle is too low to be detected by the laserscanner, and a collision occurs. The

robot must be able to recover from the crash and keep on navigating towards the goal with an updated global path in this scenario. The Battery_1 scenario tests if the robot tries to navigate toward a distant goal with a low battery charge. The simulated battery is emptied to a low level, and a new navigation goal is set. If the battery charge runs below zero percent during the navigation, the test fails.

5.2 Results

The results in table 5.3 show an increase in the successful handling of the test scenarios for most test cases. The system supervision component is very reliable and enables the system to recover from an unexpected sensor failure. The more advanced behaviors implemented can improve the autonomous handling of the scenarios. The Battery Scenario was successful every time it was induced. However, the path planning behavior for finding alternative goals was unsuccessful about 50 percent of the time. The test showed that alternative methods for finding new goals would probably have resulted in better performance. The collision behavior during driving worked well, and the map updates are an essential feature for the success of the scenarios. Nevertheless, there is a lack of intelligence and autonomy when the robot stands still. A collision is forced, but the implemented behavior relies on the same mechanism as during the driving scenario. The missing differentiation led to a test case where the robot reversed into a wall because it was turned 90 degrees by the collision. That test failed because a human operator would be needed to correct the robot. This lack of robustness was not apparent in the driving collision scenario because the robot's force was insufficient to cause a significant shift in orientation when colliding with obstacles.

The architectural design choices meet all the non-functional requirements. Implementing the whole behavior planning and system supervision system eliminates a single point of failure for the system. Even when the Navigation2 system collapses, is the robot able to move to a degree and, more importantly, stop completely (compare table ??, non_fn_req). The behavior tree and algorithms implemented in the behavior tree are all deterministic (non_fn_req3). The robots' behavior goes beyond pure reactivity when navigating through environments, as the advanced behaviors use a dedicated planning phase before the actions are carried out (non_fn_req4). Finally, the system's performance when checking the conditions of the BT is below the desired threshold of 10ms, as the whole system runs in multiple threads, which allows the behavior tree to complete one complete execution cycle in about 5ms with the current system. (non_fn_req2).

Table 5.1: Sensor Scenarios

Name	Related Requirements	Description	Success Criteria
Lidar_1	fn_req1, fn_req2, fn_req3, fn_req4, fn_req5	Robot is standing still, Lidar node crashes	Reset the system
Lidar_2	fn_req1, fn_req2, fn_req3, fn_req4, fn_req5	Navigating with 0.25 m/s straight towards goal (1m away)	Reset system, reach goal
Lidar_3	fn_req1, fn_req2, fn_req3, fn_req4, fn_req5	Navigating with 0.25 m/s and 0.5 rad/s towards goal (1m away)	Reset system, reach goal
IMU_1	fn_req1, fn_req2, fn_req3, fn_req4, fn_req5	Robot is standing still, IMU node crashes	Reset system
IMU_2	fn_req1, fn_req2, fn_req3, fn_req4, fn_req5	Navigating with 0.25 m/s straight towards goal (1m away)	Reset system, reach goal
IMU_3	fn_req1, fn_req2, fn_req3, fn_req4, fn_req5	Navigating with 0.25 m/s and 0.5 rad/s towards goal (1m away)	Reset system, reach goal
Odom_1	fn_req1, fn_req2, fn_req3, fn_req4, fn_req5	Robot is standing still, Odom node crashes	Reset the system
Odom_2	fn_req1, fn_req2, fn_req3, fn_req4, fn_req5	Navigating with 0.25 m/s straight towards goal (1m away)	Reset system, reach goal
Odom_3	fn_req1, fn_req2, fn_req3, fn_req4, fn_req5	Navigating with 0.25 m/s and 0.5 rad/s towards goal (1m away)	Reset system, reach goal

Table 5.2: Behavior Scenarios

Name	Related Requirements	Description	Success Criteria
NoPath Found_1	fn_req7	Path to goal can not be calculated, robot is standing still	Reset system, calculate path to goal, reach goal
NoPath Found_2	fn_req7, fn_req8	Goal is unreachable, robot is standing still	Alternative goals, close to the original are tested to be reachable
Collision_1	fn_req2, fn_req4, fn_req6	Robot is standing still and a collision with the robot is caused	Robot can get out of collision state, navigation to goals still working
Collision_2	fn_req2, fn_req4, fn_req6	The robot is driving and collides with an undetected obstacle (0.25m/s)	Robot can get out of collision state, navigation to goals still working, undetected obstacle gets added to map
Battery_1	fn_req9	The robot battery runs low	The robot will not drive to a goal that is outside of its reachable range
Motor_1	fn_req2, fn_req5	Hardware failure on the motors	n.a.

Table 5.3: Results

Name	Number of Runs	Percentage Successful Without Behavior Planning	Percentage Successful With Behavior Planning
Lidar_1	5	0	100
Lidar_2	5	0	100
Lidar_3	5	0	100
IMU_1	5	0	100
IMU_2	5	0	100
IMU_3	5	0	100
Odom_1	5	0	100
Odom_2	5	0	100
Odom_3	5	0	100
NoPath Found_1	20	0	100
NoPath Found_2	20	0	55
Collision_1	20	0	95
Collision_2	20	0	100
Battery_1	10	0	100
Motor_1	10	0	0

Chapter 6

Conclusion

This thesis demonstrated how to incorporate advanced behavior planning in ROS2 mobile robots. The goal was to achieve higher levels of autonomy and robustness by reducing the number of scenarios in which a human operator is needed. Implementing the system supervisor component improves the system's robustness and provides a safe basis for executing more intelligent behaviors. The implemented behaviors to react to sensor failures and robot collisions are valuable improvements to the system and improve current ROS2 mobile robots significantly. The layout of the behavior tree is structured to allow more complex behaviors to be safely executed because possible scenarios that interfere with a correct environment representation are checked and dealt with before complex behaviors get executed based on that representation. The implemented layout choice for the tree is not mentioned in the literature. However, it has proven to be a sensible approach to model robot behavior safely and robustly.

Furthermore, the system architecture is a good base for further development without facing problems or limitations in the later stages of developing behaviors. The architecture shows a novel way that works with the asynchronous ROS2 executor properties and allows fast execution of both the behavior planner and the system components.

Nevertheless, the behavior tree design could be modified to use blackboard values more effectively to increase the information flow between action nodes in the tree. Also, the tree does not utilize the reactive control flow nodes, and no action node in the tree is designed to allow multiple threads at the current state. Incorporating so-called stateful nodes into the tree's structure to allow co-routines would increase the possibility of creating better and more deliberative behaviors.

Chapter 7

Outlook

The implemented behaviors are a first step to increasing autonomy, but the behavior planner needs to create many more behaviors and fallbacks. Some behaviors not implemented in this thesis incorporate the Intel RSS rules for autonomous driving, like keeping appropriate distances, right-of-way rules, and adaptable speed based on occlusions. Another area of future work is the creation of dedicated behaviors for multi-robot systems. E.g., sharing sensor data with other robots to overcome complete sensor failures or intelligent path planning to avoid collisions when robots can access the planned paths of other robots. Also, more work is needed to define a robot's safe state or position after triggering an emergency stop. Reaching these minimum risk states after emergencies is then the duty of the behavior planner.

The behaviors must be more deliberative and robust to achieve more independence from human operators. One key question in autonomous driving is the responsibility of a vehicle in a crash because, with higher levels of automation, the human is not needed anymore. However, a machine can not be responsible for its action because it was programmed to do so. Nevertheless, intelligent, robust, and independent decision-making almost always involves machine learning which makes decisions based on training with dedicated datasets and not explicitly written lines of code. Because a system in which the outcome for a high-risk scenario is unclear, machine learning behaviors may not be suited to be used independently. Therefore, an important work area is incorporating machine learning algorithms into a deterministic system, like a behavior tree. A behavior tree could execute and monitor the behaviors to safeguard the indeterministic behaviors and override them if necessary.

Lastly, the results must be reproduced on an actual mobile robot. The layout of the behavior tree will stay the same, but further effort is needed to adapt the action nodes of the tree for a real system.

Bibliography

- [1] E. F. und Innovation (EFI), “Gutachten zu forschung, innovation und technologischer leistungsfaehigkeit deutschlands 2018,” EFI, Berlin, Tech. Rep., 2018.
- [2] SAEInternational, *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, apr 2021. [Online]. Available: https://doi.org/10.4271/J3016_202104
- [3] R. Murphy, R. Murphy, and R. Arkin, *Introduction to AI Robotics*, ser. A Bradford book. MIT Press, 2000. [Online]. Available: <https://books.google.de/books?id=RVlnLX6FrwC>
- [4] R. Brooks, “A robust layered control system for a mobile robot,” *IEEE Journal on Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.
- [5] G. Velasco-Hernandez, D. J. Yeong, J. Barry, and J. Walsh, “Autonomous driving architectures, perception and data fusion: A review,” in *2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP)*, 2020, pp. 315–321.
- [6] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Matheis, “A self-driving car architecture in ros2,” in *2020 International SAUPEC/RobMech/PRASA Conference*. IEEE, 2020, pp. 1–6.
- [7] M. Zimmermann and F. Wotawa, “An adaptive system for autonomous driving,” *Software Quality Journal*, vol. 28, no. 3, pp. 1189–1212, 2020.
- [8] A. Bacha, C. Bauman, R. Faruque, M. Fleming, C. Terwelp, C. Reinholtz, D. Hong, A. Wicks, T. Alberi, D. Anderson *et al.*, “Odin: Team victortango’s entry in the darpa urban challenge,” *Journal of field Robotics*, vol. 25, no. 8, pp. 467–492, 2008.
- [9] L. De Silva and H. Ekanayake, “Behavior-based robotics and the reactive paradigm a survey,” in *2008 11th International Conference on Computer and Information Technology*, 2008, pp. 36–43.
- [10] R. Arkin, R. Arkin, and R. Arkin, *Behavior-based Robotics*, ser. Bradford book. MIT Press, 1998. [Online]. Available: <https://books.google.de/books?id=mRWT6alZt9oC>
- [11] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, *Modeling Software with Finite State Machines: A Practical Approach*, 05 2006.

- [12] M. Foukarakis, A. Leonidis, M. Antona, and C. Stephanidis, “Combining finite state machine and decision-making tools for adaptable robot behavior,” in *Universal Access in Human-Computer Interaction. Aging and Assistive Environments*, C. Stephanidis and M. Antona, Eds. Cham: Springer International Publishing, 2014, pp. 625–635.
- [13] J. Ziegler, P. Bender, M. Schreiber, H. Lategahn, T. Strauss, C. Stiller, T. Dang, U. Franke, N. Appenrodt, C. G. Keller, E. Kaus, R. G. Herrtwich, C. Rabe, D. Pfeiffer, F. Lindner, F. Stein, F. Erbs, M. Enzweiler, C. Knöppel, J. Hipp, M. Haueis, M. Trepte, C. Brenk, A. Tamke, M. Ghanaat, M. Braun, A. Joos, H. Fritz, H. Mock, M. Hein, and E. Zeeb, “Making bertha drive—an autonomous journey on a historic route,” *IEEE Intelligent Transportation Systems Magazine*, vol. 6, no. 2, pp. 8–20, 2014.
- [14] D. C. Conner and J. Willis, “Flexible navigation: Finite state machine-based integrated navigation and control for ros enabled robots,” in *SoutheastCon 2017*, 2017, pp. 1–8.
- [15] H.-W. Park, A. Ramezani, and J. W. Grizzle, “A finite-state machine for accommodating unexpected large ground-height variations in bipedal robot walking,” *IEEE Transactions on Robotics*, vol. 29, no. 2, pp. 331–345, 2013.
- [16] G. Florez-Puga, M. A. Gomez-Martin, P. P. Gomez-Martin, B. Diaz-Agudo, and P. A. Gonzalez-Calero, “Query-enabled behavior trees,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 4, pp. 298–308, 2009.
- [17] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, “A survey of behavior trees in robotics and ai,” *Robotics and Autonomous Systems*, vol. 154, p. 104096, 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0921889022000513>
- [18] S. Feyzabadi and S. Carpin, “Risk-aware path planning using hierarchical constrained markov decision processes,” vol. 2014, 08 2014.
- [19] A. M. Andrew, “Reinforcement learning: An introduction by richard s. sutton and andrew g. barto, adaptive computation and machine learning series, mit press (bradford book), cambridge, mass., 1998, xviii+ 322 pp, isbn 0-262-19398-1,(hardback,£ 31.95).” *Robotica*, vol. 17, no. 2, pp. 229–235, 1999.
- [20] S. Tanwar. (2019) Markov chains and markov decision process. [Online]. Available: <https://sanchittanwar75.medium.com/markov-chains-and-markov-decision-process-e91cda7fa8f2>
- [21] V. Krishnamurthy, *Partially Observed Markov Decision Processes*. Cambridge university press, 2016.
- [22] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.
- [23] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science*

- Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: <https://www.science.org/doi/abs/10.1126/scirobotics.abm6074>
- [24] S. Macenski, F. Martin, R. White, and J. Ginés Clavero, “The marathon 2: A navigation system,” in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.
- [25] S. J. Konrad, *Defining and using requirements patterns for embedded systems*. Michigan State University, 2003.