Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Titel der Bachelorarbeit oder Titel der Masterarbeit oder Titel der Diplomarbeit

---

Bachelorarbeit

Name des Studiengangs
Ingenieurinformatik

Fachbereich 2

vorgelegt von
Lukas Evers

Datum: 23.09.2022

Erstgutachter/in: Prof. Dr. Frank Burghardt
Zweitgutachter/in: Dr. Ahmed Hussein

Abstract

List of Figures

2.1 SAE Levels Of Driving Automation [1] 4
2.2 Common Autonomous Driving Architecture [2] [3] 5
2.3 Hybrid Planning Architecture [4] 8
2.4 Example of a finite state machine transition diagram [5] 9
2.5 Example of a behavior tree [6] 11

3.1 Navigation2 Architecture [7] 15
3.2 Navigation2 Behavior Tree 15
3.3 Turtlebot3 model in Gazebo simulator 17

4.1 Component Diagram of System Architecture 22
4.2 Lidar Fallback 24
4.3 Collision Fallback Behavior 25
4.4 Battery Fallback Behavior 26
4.5 Path Planning Fallback 27

List of Tables

2.1 The five types of nodes [6] 11

3.1 ROS Communication Options 14
3.2 Functional Requirements 18
3.3 Non-Functional Requirements 18
3.4 Control Nodes in BT.CPP 19
3.5 Decorator Nodes in BT.CPP 20

4.1 Types of saved data 22

Contents

Abstract **i**

1 Introduction **1**

2 State of the Art **3**
- 2.1 Automated and Autonomous Vehicles . . . 3
- 2.2 Autonomous Driving Navigation Architectures . . . 3
- 2.3 Behavior Types . . . 6
  - 2.3.1 Reactive . . . 6
  - 2.3.2 Deliberative . . . 6
  - 2.3.3 Hybrid . . . 7
- 2.4 Behavior Planning Approaches . . . 9
  - 2.4.1 Finite State Machines . . . 9
  - 2.4.2 Behavior Trees . . . 10
  - 2.4.3 POMPDs . . . 11

3 Concept **13**
- 3.1 Current State . . . 13
  - 3.1.1 ROS . . . 13
  - 3.1.2 Navigation2 . . . 13
  - 3.1.3 Turtlebot3 . . . 16
- 3.2 Requirements . . . 16
- 3.3 Risk Analysis For Requirement Prioritization . . . 17
- 3.4 System Solution . . . 17
- 3.5 Software and Simulation . . . 19
  - 3.5.1 Behaviortree.CPP . . . 19
  - 3.5.2 Groot . . . 19
  - 3.5.3 Gazebo Simulation Environment . . . 20

4 Implementation **21**
- 4.1 System Architecture . . . 21
- 4.2 Data Backup . . . 21
- 4.3 Command Velocity Decision Gate . . . 22
- 4.4 Behavior Tree Structure . . . 22
  - 4.4.1 BT System Supervision . . . 23

4.4.2 BT Advanced Behaviors . . . 23

5 Evaluation **29**
5.1 Scenarios . . . 29

6 Summary **31**
6.1 Conclusion . . . 31
6.2 Prospect . . . 31

Chapter 1

Introduction

Research question: **How can behavior planning increase the robustness and autonomy of a robot running ROS2?**

Core Problem - too little autonomy in a standard autonomous mobile robot running ROS navigation(AMR)

Description of the Problem:

Despite the advancements in autonomous driving the robot regurlarly needs an operator in case smth unexpected happens.

This may include: unexpected collisions, a system restart is needed, battery is empty, the environment is too crowded and the robot gets stuck

The system is not able to react and handle unforeseen situations.

This can lead to safety issues during the operation of a robot. These safety issues can lead to uncontrolled driving maneuvers due to wrong sensor data or a wrong environment represantation.

Due to the need of the robot to require an operator for safe, the robot is not really autonomous despite it being labeled as such.

(Difference is defined by the SAE Levels for automonous driving in cars.)

Relevance, need for research:

It is important for the robot to possess the ability to navigate in an uncertain environment to guarantee the safety of the robot itself and all other actors in its environment, these can be other robots and humans.

In theory the robot can perform fully autonomous navigation including mapping and localization but as soon as something out of the ordinary happens the robots autonomy is not guaranteed to be reliable anymore. Default robot is not able to represent all possible fail states and does not detect failures on a systematic level, but only inside its navigation related subsystem. And even when failures inside this subsystems are discovered, the options to handle these problems are very limited and often do not deal with the problems in an autonomous way. The reliance on human-needed problem solving decreases the robots usefulness when tasked with real goals.

Having a robust and safe robot behaviour opens the door to many applications in a more diverse set of challenging environments. Usually a reinforcement learning approach with broad training data set is a great way to handle a multitude of unknown environments and situations, but one of the biggest problems is the lacking determinism in such systems.

In the robotic world exists a need for a deterministic system which can enhance and override the decision making process and navigation capabilities of the robot.

Planned solution:

This thesis will explore an approach how an extension of the current software architecture and behavior planning capabilities for an autonomous mobile robot that is running ROS2 can improve the autonomy of the robot.

Expected results:

The desired outcomes of these implemented extension will be an increase of robustness and decrease of required human actions during a number of scenarios. In these scenarios, failures and and problems are artificially induced to test the robots abilities to behave autonomously.

Research gap is not named so far

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 what qualifies as an improvement in the autonomous behaviour of a mobile robot, one needs to look at the defintion of automated and autonomy. The EFI defines the term autonomy in the context of robotics as a system which can act without human instructions and still solve complex tasks, make decisions, learn independently aswell as react to unforeseen circumstances [8]. The definition specifies the needed requirements to make a robot fully autonomous. The automotive industry defines vehicle autonomy in levels based on the human driver's need for supervision and possible intervention during the execution of complex driving tasks [1]. Figure xx depicts the different levels of autonomy of passenger cars on a scale between no automation to completely autonomous.

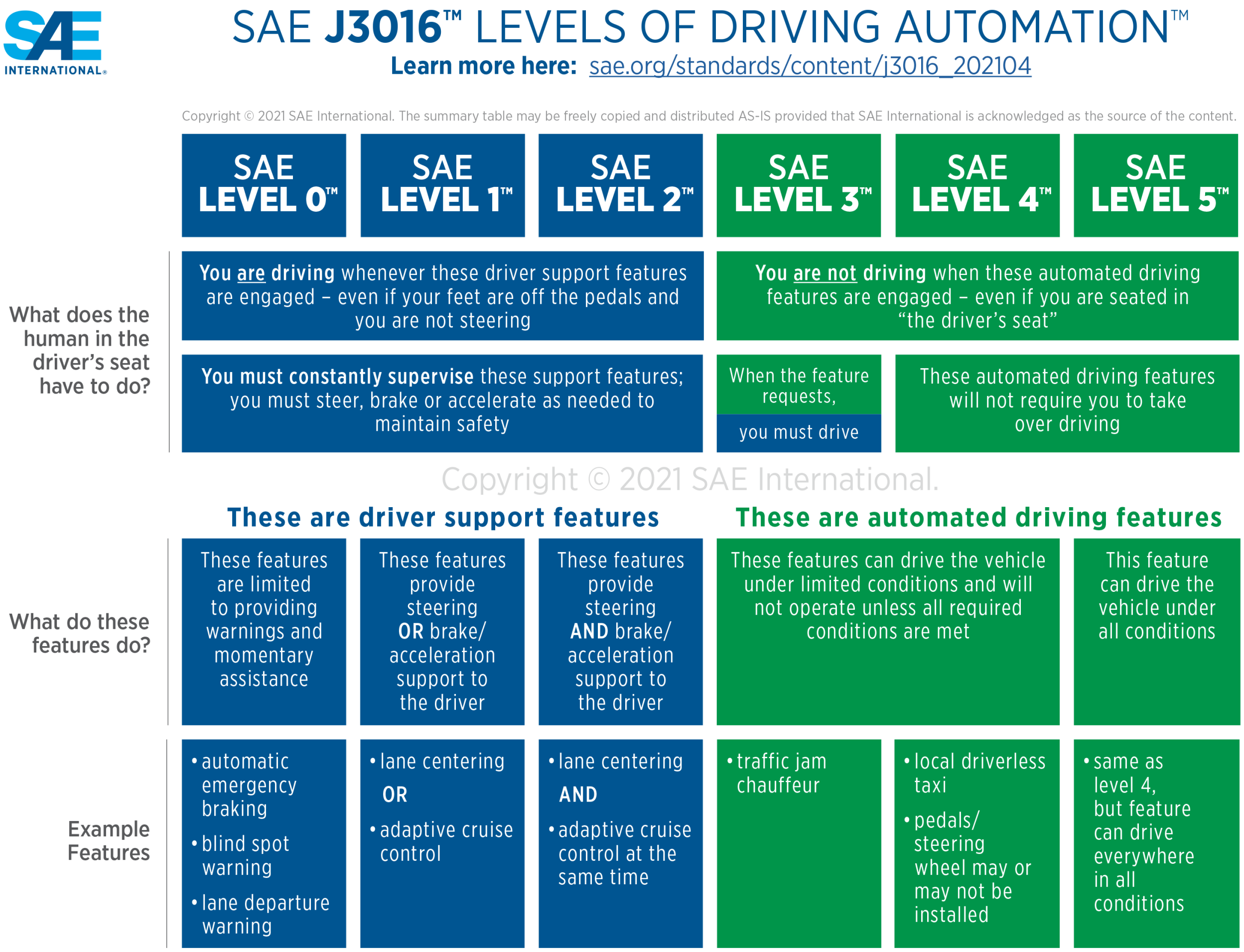
Both definitions entail the reliance on human supervision and guidance as the central part of what hinders a system or vehicle to become autonomous. By decreasing the instances a human intervention during the operation of a robot, one can increase the automation level towards full autonomy. But this has to be done by equipping the robot with robust and context-driven decision-making and planning capabilities, otherwise a robot which would never need a human to operate could easily be created. But this robot could not be considered autonomous as it would not able to solve complex tasks and make intelligent decisions.

Different levels of autonomy require different methodologies to achieve them.

SAE Levels EFI Gutachten Fachforum

2.2 Autonomous Driving Navigation Architectures

To gain a better understanding how behavior planning influences the autonomy of a robot, one can take a look at the latest and most used software architectures for autonomous driving on a functional level. Most of the current autonomous driving architectures are implementing a hierarchical structure that follows the "Sense - Think - Act" paradigm [9]. As shown in figure xx 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



SAE **J3016**™ LEVELS OF DRIVING AUTOMATION™

Learn more here: sae.org/standards/content/j3016_202104

Copyright © 2021 SAE International. The summary table may be freely copied and distributed AS-IS provided that SAE International is acknowledged as the source of the content.

	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 are not driving** when these automated driving features are engaged – even if you are seated in "the driver's seat"		
	You must constantly supervise these support features; you must steer, brake or accelerate as needed to maintain safety			When the feature requests, you must drive	These automated driving features will not require you to take over driving	
	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	• automatic emergency braking • blind spot warning • lane departure warning	• lane centering **OR** • adaptive cruise control	• lane centering **AND** • adaptive cruise control at the same time	• traffic jam chauffeur	• local driverless taxi • pedals/ steering wheel may or may not be installed	• same as level 4, but feature can drive everywhere in all conditions

Copyright © 2021 SAE International.

Figure 2.1: SAE Levels Of Driving Automation [1]

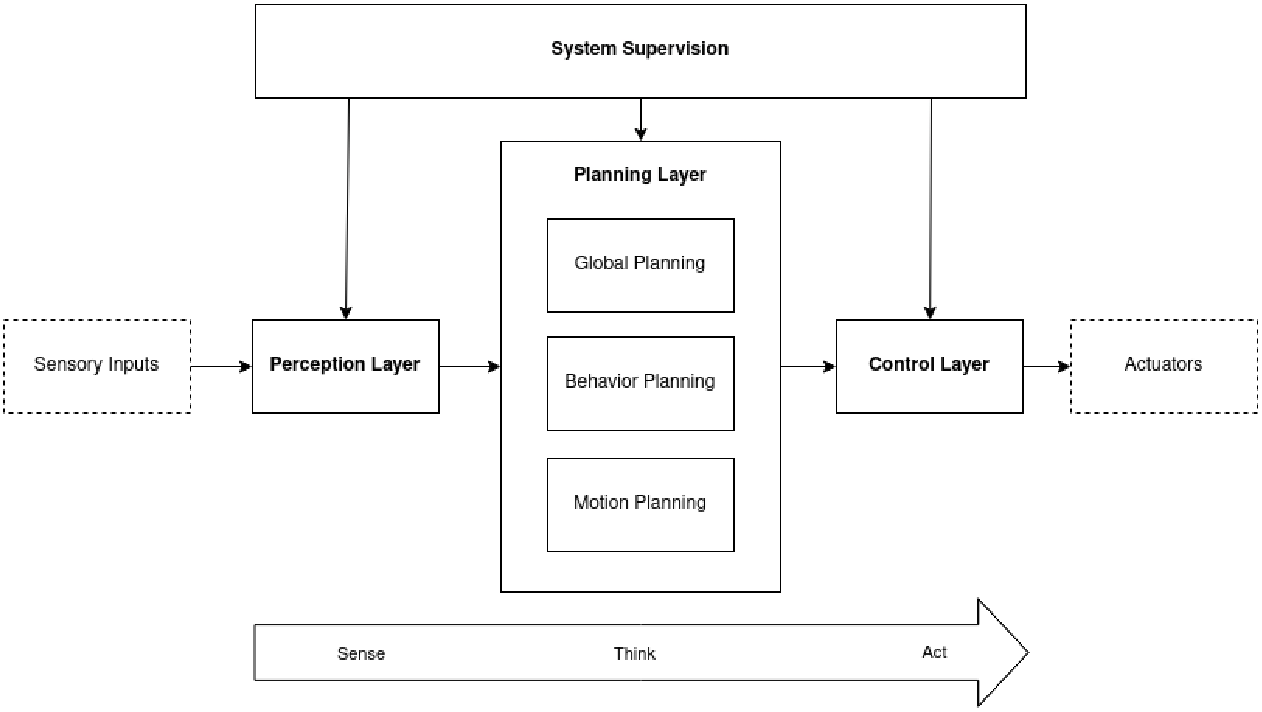


Figure 2.2: Common Autonomous Driving Architecture [2] [3]

to the destination and creates a path. A second planning calculation is triggered to compute the motion commands with the constraints of the system (e.g. size, turning radius, etc.). These motion commands then get converted to control the motors. The planning sequence can be further divided into global planning, behavior planning and local planning. Global planning,also named route planning, is responsible for outputting a path from start to goal. This is comparable to a person using Google Maps to find the shortest or fastest path to a far away city. In this analogy, the behavior planner is responsible to follow the traffic rules on the trip to the destination, e.g. stopping at red lights, giving right of way to other vehicles, following the speed limit. The local planning, also named motion planning, module takes the input from the behavior layer and has the task to let the vehicle drive in accordance to the executed behavior, so that it stays in its lane and comes to a stop at a red light [10].

Due to safety considerations, many autonomous driving architectures implement an additional system supervision layer which monitors the execution of the other system components. This supervisor checks the health of all software components. In case one or more components fail to pass the health check the supervisor is responsible for ensuring that the system does not continue the normal driving routine. The supervisor triggers measures to restore the normal functionality or if that is not possible bring the robot into a safe state (zimmermann2020).

Considering that with these architectures, vehicles can achieve an SAE level of autonomy up to level 4 (bacha 2008), 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 levels of autonomy are achievable through the expansion of the behavior planning module inside of the planning layer.

bacha2008 reke 2020 wei2014 brooks1986 dhillon 2002 gonzales2016

2.3 Behavior Types

The behavior planning module contains a set of behaviors to operate in a environment. The task of the module is then to choose the best behavior to execute. In this sense a behavior is defined as a mapping of sensory inputs to a pattern of actions that in turn carry out a specific task [9]. A good behavior planning approach provides the robot with adequate 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 execute in order to achieve a higher level goal.

2.3.1 Reactive

During the emergence of behavior based robots, the first type of behavior that was implemented in many robots was a simple reactive behavior. This behavior maps a sensory input directly to motor commands. In human behavior this type of behavior resembles reflexes, like the 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 [11]. Reactive behaviors can be chained together to result in 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 behaviours 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 these kind of behaviours useful for system supervisor, where sometimes immediate reactions with low latency are required to ensure vehicle safety. But a system with a reactive behavior planning approach will always fail to meet the requirements for higher levels of autonomy due to the fact. This does not mean that reactive behaviors can not be part in highly autonomous systems as 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 enable complex behaviors. In addition, the creation of sequences of reactive behaviors that produce the target behavior is a complicated task [9]. Behaviors that involve a planning and decision making aspect are defined as deliberative. Unlike reactive behaviors, deliberative behaviors are following the Sense, Think, Act paradigm. They are not mapping sensory inputs directly into motor commands. Instead deliberative-type behaviors are deciding the best course of action before acting and only then proceed with the exution. The use of deliberative behaviors in the planning module is what enables a robot to meet the defintion of autonomy, e.g. making decisions and react to unforeseen circumstances. So in order to improve the autonomy of a robot, the focus should be on the creation of deliberative behaviors. A behaviour planner with a deliberative approach makes decision based on current sensor data, as well as previously processed data. By incorporating older sensor information into the decision making process, the deliberative behavior planner can make predictions about the changes that will happen in the environment. This 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 the motion planning considerably. This is due to a better understanding of how obstacles are interacting with the planned path of the robot in relation to the time at which the robot and obstacle are predicted to intersect paths. A purely reactive planner could never determine if an obstacle is destined to intersect the robots path in the future and would reroute constantly in order to accomplish the goal. A deliberative planner could 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 to intersection point. Other actions in scenario are possibly slowing down or speeding up briefly to avoid an obstacle and staying on the current path as this is calculated to be quicker than rerouting and taking a longer path. Generally, these cognitively more complex behaviours mimic human skills and thus make a system more autonomous.

Talk about machine learning as deliberative behavior -¿ true autonomy bc learning and unknown scenarios, programmers can not cover every scenario, so a well designed/trained ai is needed to make a system really autonomous, but -¿ missing determinism and unaccountability of machines, hence we need behaviors that are still hard coded and deterministic, these can be highly automated. traditional behaviour and ai generated behaviours can and must coexist in the system (adler2019)

silva2008 ingrand 2014 vasiolopolous 2022

2.3.3 Hybrid

To create reliable aswell as intelligent systems one can combine reactive and deliberative behaviours in a hybrid model. This behavior planning module is then able to quickly react to incoming sensor data and still exhibit high levels of automation.

Figure xx shows the areas where reactive and deliberative behaviours have their 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 adressing these points, deliberative robots are very focussed on a small problem domain and are not robust [4]. The incorporation of reactive behaviors into the behavior planning combats this problem. In this multi-layer approach the higher level, more deliberative planners are able to override the reactive planners in order to allow the system to be more flexible and adaptible but still maintain fast reaction times. This architecture leads to more robust robots that are able to be deployed in a wider variety of environments, thus leading to higher levels of autonomy.

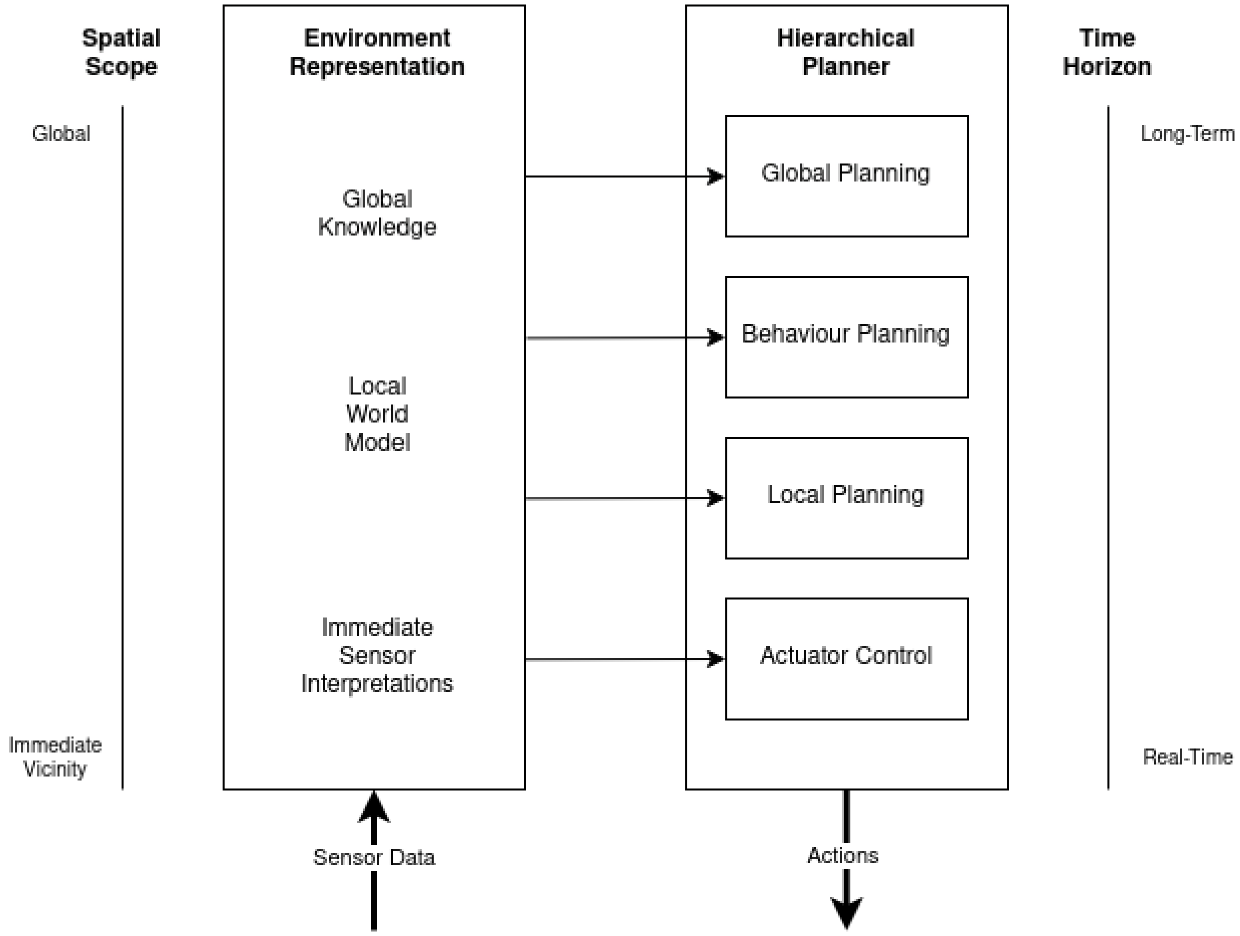

Spatial Scope
Global
Immediate Vicinity
Environment Representation
Global Knowledge
Local World Model
Immediate Sensor Interpretations
Hierarchical Planner
Global Planning
Behaviour Planning
Local Planning
Actuator Control
Time Horizon
Long-Term
Real-Time
Sensor Data
Actions

Figure 2.3: Hybrid Planning Architecture [4]

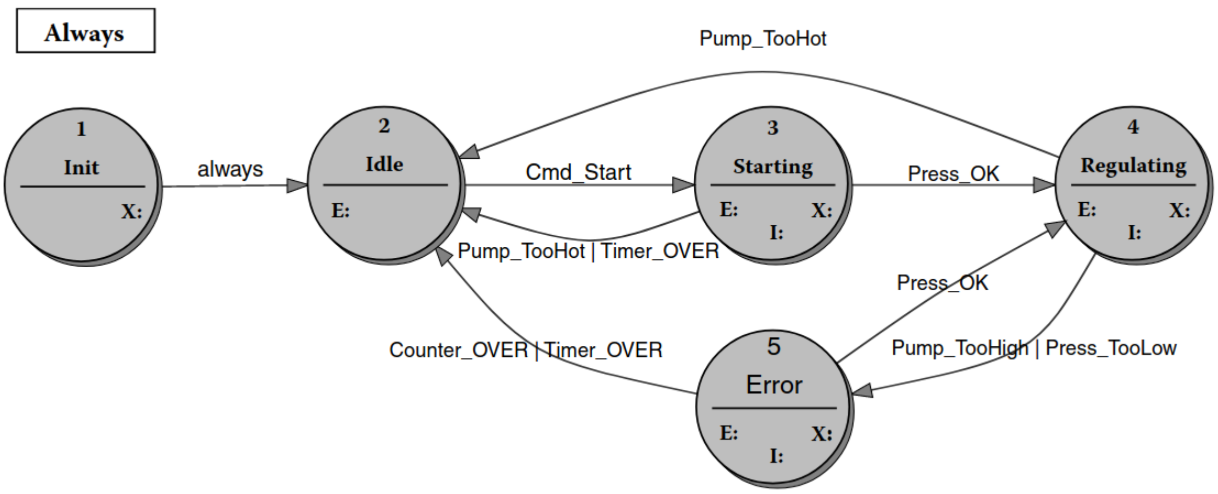


Figure 2.4: Example of a finite state machine transition diagram [5]

2.4 Behavior Planning Approaches

Robots that use a hybrid, hierarchical behavior planning approach need a system which decides on a high level which behaviors are to be executed. This allows overriding simple reactive behaviors with more complex, deliberative behaviors when the system benefits from it.

The different solution for such behavior execution system all share the fact that they make use of states a robot can be in. Based upon a robots state and additional information about the environment, different strategies and behaviors are executed during the runtime of the system.

2.4.1 Finite State Machines

Finite State Machines (FSM) are commonly used in the environment of autonomous driving. They are used to describe behavior in a form of states, which in turn execute action. A finite state machine is defined by a list of possible states S, a starting state s0, a set of state transitions delta, a set of final states F and the input alphabet sigma. At any given time only a single state is selected and its containing actions are executed.

Figure xx depicts an example state machine with different states and examplatory 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 a good way to understand the system behavior but can not show the details of the modedeled behavior, like the actions [5].

Creating small state machines can easily and quickly be accomplished through the use of various feature-rich and performant libraries [12]. Finite state machines are often being used for comprehensive modeling of reactive behavior and sequential behaviors in autonomous driving functions [13]. These finite state machines make use of nested state machines inside of another state machine. This reduces the complexity of the state machine as the transition do not need to be modeled, because the start and final state of a nested state machine is treated as just one state inside of the larger state machine. Despite this possibility to simplify state machines, as a modeled behavior system grows larger to accomodate more complex behaviors into the state machine the effort to expand and maintain the state machine grows rapidly. This is due to the modeling effort 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 suited very well suited for fast, reactive type behaviors and still allows a hierarchical approach to behavior planning.

allgeuer 2013 connor 2017 ziegler 2014

2.4.2 Behavior Trees

Behavior Trees (BT) are another way to model and control the behavior of autonomous systems. Behavior Trees first found big acceptance in the computer game industry where they are mainly used to model artificial intelligence for non-player characters [16]. Every tree consists of one root and many child, parent and leaf nodes. Leaf nodes are also called execution nodes, while non-leaf nodes are called control or control flow nodes. The execution of a behavior tree is done by ticking the root node of tree. This signal then travels down to the child node of the root node, where either control nodes are being 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 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” of ”running”, and only return ”success” itself when every child node was successfully ticked. If any of the child 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 will tick its child nodes as long as they return ”failure” or ”running” and will return ”success” if one of the ticked node returned ”success”. It will only return ”failure” if all of the child nodes ”returned success”.

Figure xx depicts an example of a BT with multiple levels and action nodes. The nodes with the arrow (-¿) symbol are sequence nodes, the question mark (?) symbol denotes fallback nodes. The condition node are depicted as white ovals. The action nodes are represented as gray rectangles in the BT example.

One of the core differences to Finite State Machine is that transitions between states are not distributed across all the states but they are organized in a hierarchical tree, where the

Table 2.1: The five types of nodes [6]

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 ¿= 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

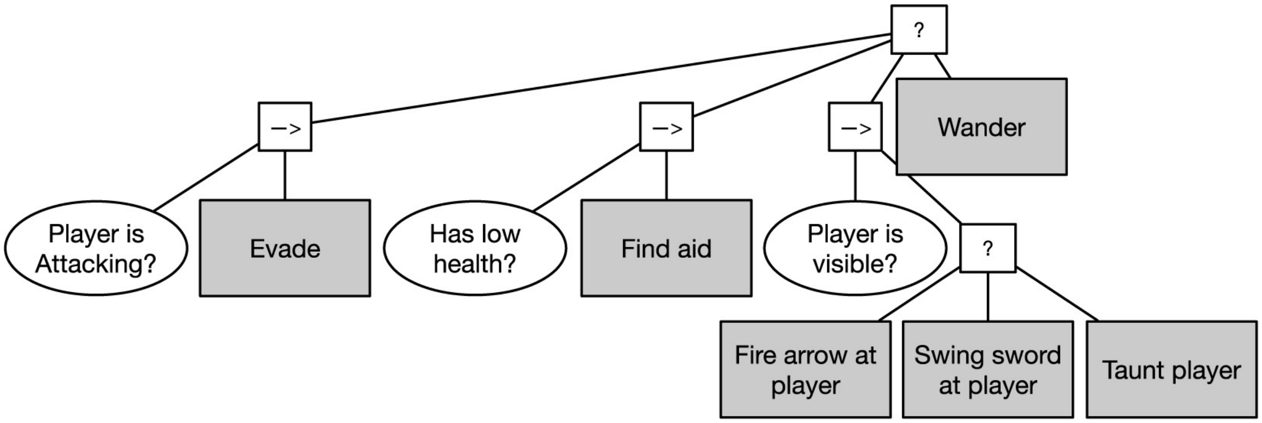

?
–>
–>
–>
Wander
Player is Attacking?
Evade
Has low health?
Find aid
Player is visible?
?
Fire arrow at player
Swing sword at player
Taunt player

Figure 2.5: Example of a behavior tree [6]

leaves are representing the states [6]. While both FSM and BT are capable of producing the same behavior in robots, the fundamental shift in how the two system are created lead to significant advantages in the modularity, synthesis and analysis of the systems at hand. These effects are more significant with an increase in size of the system. BT offer a lot more flexibility when creating an advanced behavior layer for an autonomous system.

iovino2022 marzinotto 2014 colledanchise 2018

2.4.3 POMPDs

Chapter 3

Concept

3.1 Current State

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.

3.1.1 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 xx).

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 [17].

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 [18].

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,

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

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 [7].

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 xx.

The behavior tree depicted in figure xx 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 xx (2.2), the things that Nav2 is missing are a system-wide supervision layer and a more deliberative approach to behavior planning, as mentioned in 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

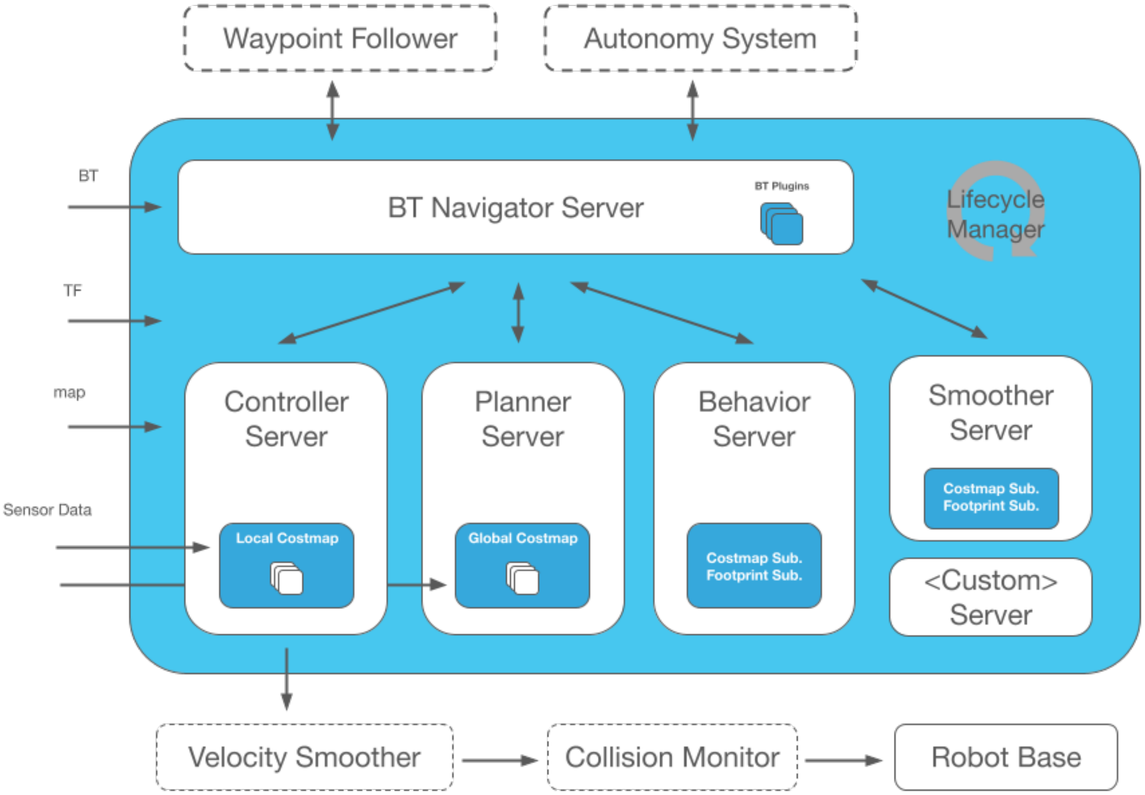

Waypoint Follower
Autonomy System
BT
TF
map
Sensor Data
BT Navigator Server
BT Plugins
Lifecycle Manager
Controller Server
Local Costmap
Planner Server
Global Costmap
Behavior Server
Costmap Sub.
Footprint Sub.
Smoother Server
Costmap Sub.
Footprint Sub.
<Custom> Server
Velocity Smoother
Collision Monitor
Robot Base

Figure 3.1: Navigation2 Architecture [7]

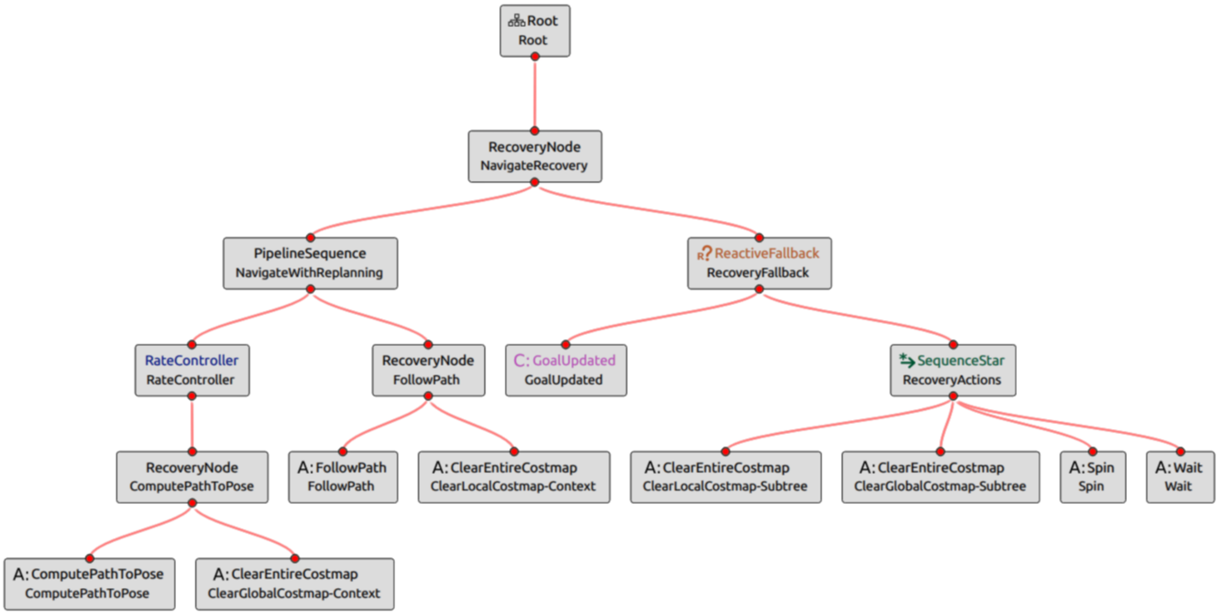

Root
Root
RecoveryNode
NavigateRecovery
PipelineSequence
NavigateWithReplanning
ReactiveFallback
RecoveryFallback
RateController
RateController
RecoveryNode
FollowPath
C: GoalUpdated
GoalUpdated
SequenceStar
RecoveryActions
RecoveryNode
ComputePathToPose
A: FollowPath
FollowPath
A: ClearEntireCostmap
ClearLocalCostmap-Context
A: ClearEntireCostmap
ClearLocalCostmap-Subtree
A: ClearEntireCostmap
ClearGlobalCostmap-Subtree
A: Spin
Spin
A: Wait
Wait
A: ComputePathToPose
ComputePathToPose
A: ClearEntireCostmap
ClearGlobalCostmap-Context

Figure 3.2: Navigation2 Behavior Tree

- 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. *Add statistics about random sensor failures every 10000h* 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.

Add a closing statement about that suitability of the Nav stack for achieving autonomy

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 xx, which enables faster development and testing for the robot. 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 xx.

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.

Do that?

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.

Non-functional requirement list

number .. Name .. Description .. Success Criteria nfreq1 Single Point of Failure The system does not have a single point of failure When parts of the system fail, the system maintains operability to a degree nfreq2 Performance The system's control loop guarantees fast reactions The average frequency with which the system checks the sensors/navigation2 is higher than 100Hz (10ms) nfreq3

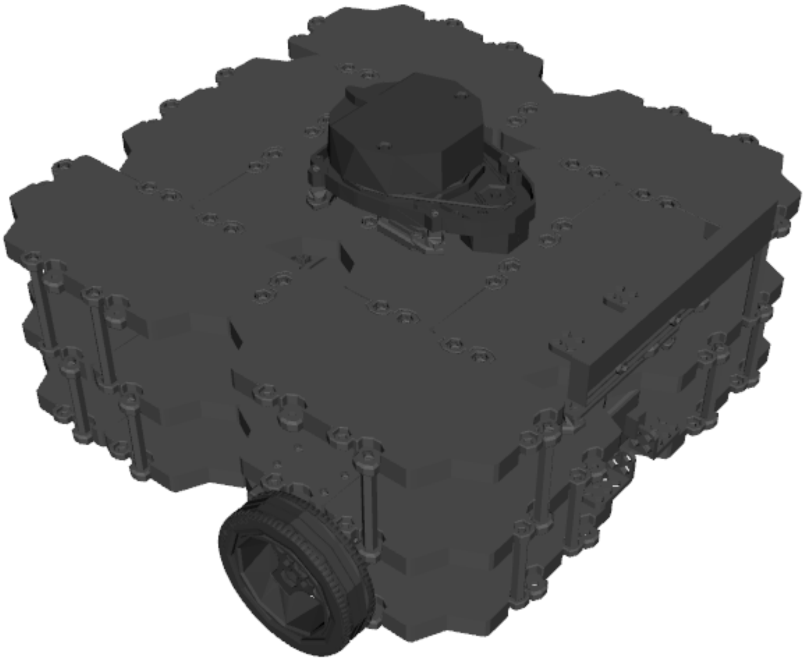


Figure 3.3: Turtlebot3 model in Gazebo simulator

3.3 Risk Analysis For Requirement Prioritization

3.4 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 broad system supervisor 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

Table 3.2: Functional Requirements

Nr.	Name	Priority	Description/Acceptance Criteria
fn_req1	Recovery	High	The system can recover from crashes. Ensure that the system can successfully reach goals despite previous crashes
fn_req2	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_req3	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_req4	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_req5	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_req6	Override Navigation2	High	The system can override navigation2. Ensure that the system's commands can always override the commands coming from navigation2.
fn_req7	Reset Goals	Medium	The system can reset and override goals set by the user so that the goal is reachable by planners.
fn_req8	Robot Range	Medium	Ensure that the robot will not run out of battery during navigation to a goal.

Table 3.3: Non-Functional Requirements

Nr.	Name	Priority	Description
nfreq1	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
nfreq2	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)
nfreq3	Determinism	High	The outcome for a given set of inputs must be deterministic, meaning that the behavior is always executed similarly.
nfreq4	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.

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 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.

3.5 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.

3.5.1 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 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 xx).

Table 3.4: 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

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 nodes and descriptions is in table xx.

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 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.

3.5.2 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

Table 3.5: 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.

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.

Screenshot of live monitoring

3.5.3 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 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 structurs, to office spaces up to complete race tracks.

Chapter 4

Implementation

4.1 System Architecture

4.2 Data Backup

The data backup component functions as a way to efficiently gather data from all components of the system and bereitstellen necessary data to the behavior tree when the planning process requires the use of past data points. This component subscribes is a 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 bestimmte length, the oldest data this data gets deleted from the array. The limited array size is twofold. On the on side, the goal is to decrease the size of the message that gets sent to the BT to limit the network auslastung 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 network load is 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 hihg priority and the execution is done quickly but it adds an additional step than just saving the data directly in the behavior tree. The reason for the outsourcing of the data backup in another component is that ROS node need an executor which spins the node constantly. The spinning allows the node to check the network if there is any work to do for the node. This means that if one of the subscribed topic received a message, the node has to execute one of its callback functions that is associated with that topic. 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 spun constantly in one thread. Have a constant spinning node would block the execution of the BT completely. If one would try to spin a node once when a node gets ticked in the BT, the problem of missing messages would become relevant because the BT might need longer than exepected to return to a node to tick it. That way the BT node that would be responsible for saving all the relevant data can not guarantee that it has picked up on all of the messages between to spins. The synchronous nature of the ROS executor spinning the nodes is the reason why the data backup is not stored within the BT.

The saved data inside of this component with the corresponding lookback time is listed in table xx.

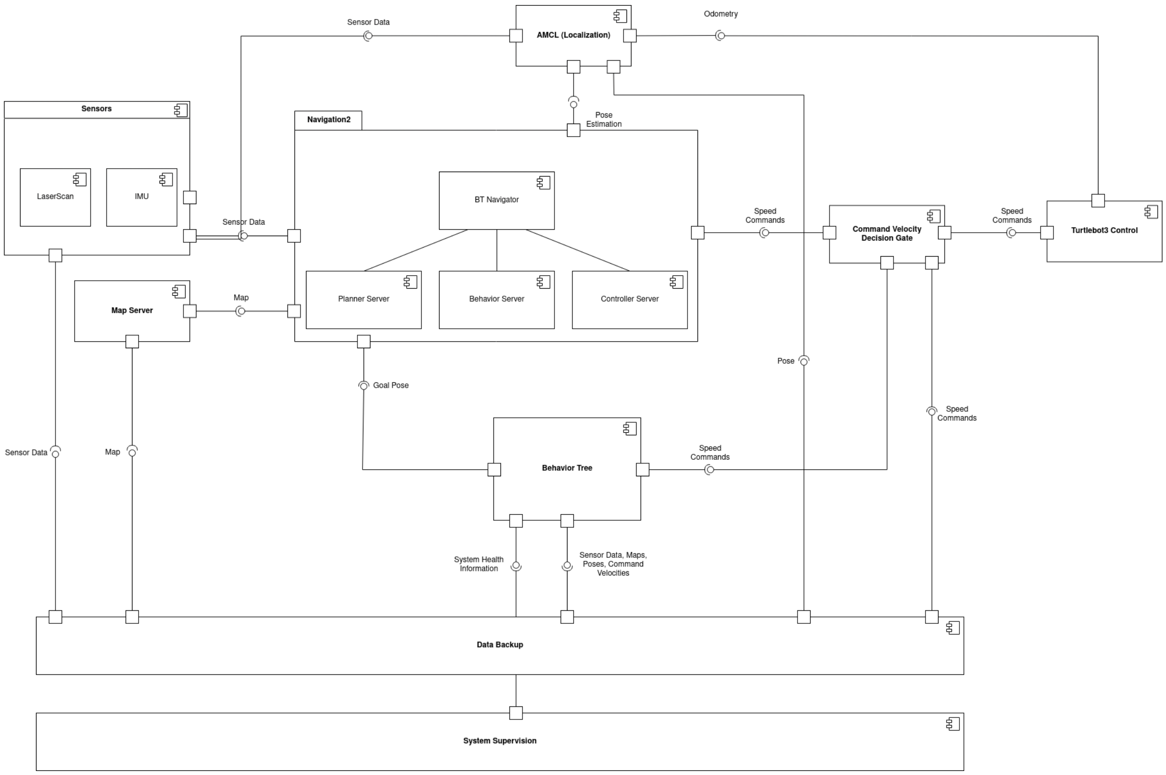

Sensor Data
AMCL (Localization)
Odometry
Sensors
LaserScan
IMU
Pose
Estimation
Navigation2
BT Navigator
Speed
Commands
Command Velocity
Decision Gate
Speed
Commands
Turtlebot3 Control
Sensor Data
Map Server
Map
Planner Server
Behavior Server
Controller Server
Goal Pose
Pose
Speed
Commands
Sensor Data
Map
Behavior Tree
Speed
Commands
System Health
Information
Sensor Data, Maps,
Poses, Command
Velocities
Data Backup
System Supervision

Figure 4.1: Component Diagram of System Architecture

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

4.4 Behavior Tree Structure

A simplified behavior tree only depicting the top level behavior control nodes is shown in figure xx. The tree receives the inital tick signal from the root in an infinite loop. The root control node is a sequence node. We made the root a simple sequence because we want the whole system to function with increasing complexity in the later parts (on the right side) of the tree. The complex behaviors require that all of the previous components and conditions are working as they should. The root sequence will not allow the execution of higher level behaviors if the system is detecting problems on more fundamental levels. The root sequence is the first step for increasing the robustness of the system to combat competing reactive and deliberative behaviors which could endanger the safety of the robot and other actors in its environments.

4.4.1 BT System Supervision

The system supervisor component is partly outsourced into a component outside of the thread of the BT, because of 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, nachfolgend also called execution checker, is receiving messages from nodes that are not implemented as ROS2 lifecycle nodes. If a node stops sending messages or stops 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 implementation of a FSM inside of the node. The current node state can be found out via a service call. However, to ensure the node is working correctly the output of the node needs to be checked too, because it might be that the FSM inside of the node did not pick up on a error. The system supervision is constantly checking the system's health. The BT is using service calls to get the health information about the compononts which are checked inside of 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. xx). In case 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 the robot will not be able to navigate in a safe manner 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 no message was received is set at one second. After that period the goal is to enable to robot to continue to make progress towards the goal but 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 the autonomy.

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.

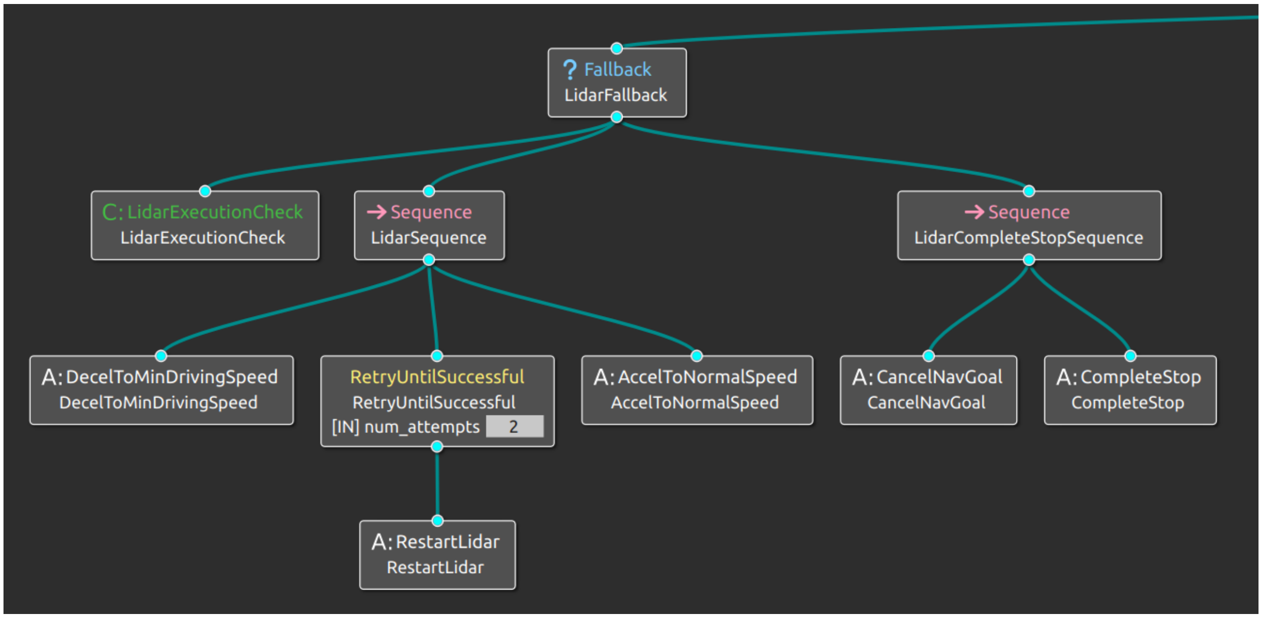

? Fallback
LidarFallback
C: LidarExecutionCheck
LidarExecutionCheck
→ Sequence
LidarSequence
→ Sequence
LidarCompleteStopSequence
A: DecelToMinDrivingSpeed
DecelToMinDrivingSpeed
RetryUntilSuccessful
RetryUntilSuccessful
[IN] num_attempts 2
A: AccelToNormalSpeed
AccelToNormalSpeed
A: CancelNavGoal
CancelNavGoal
A: CompleteStop
CompleteStop
A: RestartLidar
RestartLidar

Figure 4.2: Lidar Fallback

With a multitude of sensors covering all angles and heights and computer vision capabilities to detect obstacles collisions would be very unlikely. But even if the robot could perrfectly detect obstacles, it would still be vulnerable to outside pertubations like other robots driving into it, humans accidently touching the robot etc. . To combat this the robot must be able to first detect that a collision occured, get out of the collision state, update the map and reset all costmaps to generate a new path to the original goal.

The collision checker uses a simulated sensor to detect collisions. On a real robot the collision checker can use a IMU-based collision detection, but this has not been implemented in this behavior tree due to time constraints of this thesis. The collision checker itself is integrated into the system supervisor and the BT is making service calls to the execution checker to get the collision state of the robot periodically. The child nodes of the collision sequence make use of 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 longer 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 make sure the robot only reverses once, otherwise the robot would be driving back and forth as the data backup service would provide the BT with the commands from the last reversing action.

The modification is done in the following manner

Formula for determining execution time

Regarding the map reset, we have to determine if we are creating a new map with a SLAM method or if the map server provided a previously saved map. In case of the running SLAM, the recent updates to the map must be verwerfen to a map prior to when the collision occurred. When a static map is provided by the map server component from Nav2 the map needs to be updated with a new obstacle at the point of collision and republished by the map

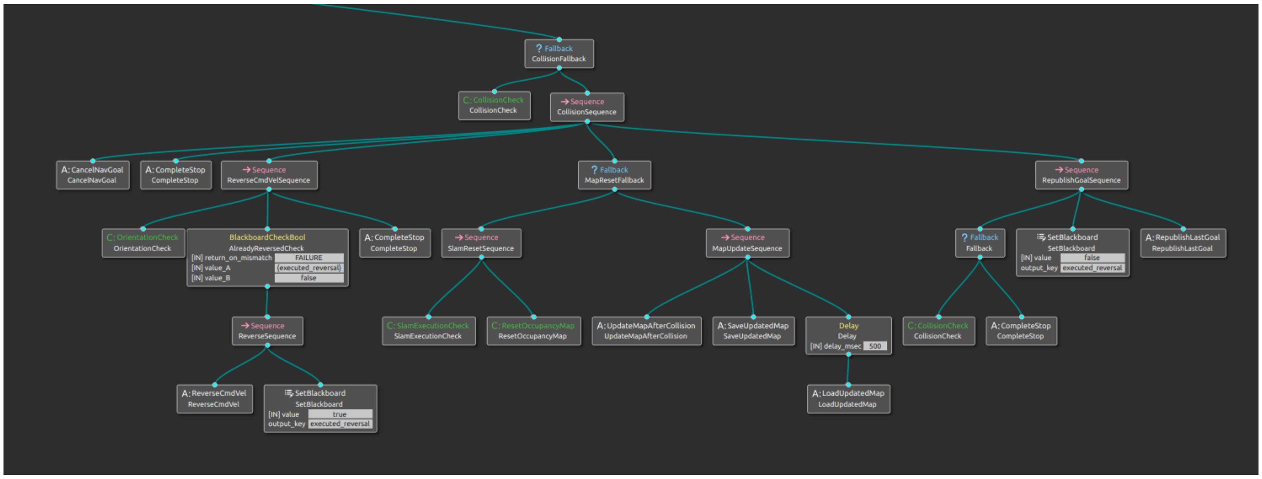

? Fallback
CollisionFallback
C: CollisionCheck
CollisionCheck
→ Sequence
CollisionSequence
A: CancelNavGoal
CancelNavGoal
A: CompleteStop
CompleteStop
→ Sequence
ReverseCmdVelSequence
? Fallback
MapResetFallback
→ Sequence
RepublishGoalSequence
C: OrientationCheck
OrientationCheck
BlackboardCheckBool
AlreadyReversedCheck
[IN] return_on_mismatch FAILURE
[IN] value_A {executed_reversal}
[IN] value_B false
A: CompleteStop
CompleteStop
→ Sequence
SlamResetSequence
→ Sequence
MapUpdateSequence
? Fallback
Fallback
SetBlackboard
SetBlackboard
[IN] value false
output_key executed_reversal
A: RepublishLastGoal
RepublishLastGoal
→ Sequence
ReverseSequence
C: SlamExecutionCheck
SlamExecutionCheck
C: ResetOccupancyMap
ResetOccupancyMap
A: UpdateMapAfterCollision
UpdateMapAfterCollision
A: SaveUpdatedMap
SaveUpdatedMap
Delay
Delay
[IN] delay_msec 500
C: CollisionCheck
CollisionCheck
A: CompleteStop
CompleteStop
A: ReverseCmdVel
ReverseCmdVel
SetBlackboard
SetBlackboard
[IN] value true
output_key executed_reversal
A: LoadUpdatedMap
LoadUpdatedMap

Figure 4.3: Collision Fallback Behavior

server.

The activity of the "MapUpdateAfterCollision" action node is depicted in figure xx. The collision point is estimated by the robot's pose which refers to the robot base link and the robot footprint. We assumed that the collision point is right in front of the robot, because the local planner favours forward drive kinematics. 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 an IMU-based collsion detection, hence we simplified the obstacle position to always be in front of the robot. The collision position is calculated with

Formula here

A 25 by 25 cm square with full occupation of that space is added to the current map and then saved to be reloaded by the map server as depicted in figure xx.

The original goal is then republished to replan the global path on the updated map.

In this manner obstacles lower than the lidar scan height can be detected. With this behavior the robot should also be able to eventually navigate around transparent obstacles which 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 to monitor the robot's battery and has to make to the decision 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.

The robot has to take into account the idle consumption, the energy to drive the motors. With this information we created a simple linear function for the energy expenditure to predict the required power depending on the length of the path. To make the algorithm fail safe, we added an additional safety factor into the function, to berücksichtigen the scenario where the robot has to reroute and drive a longer route or has to stop for a moment. The

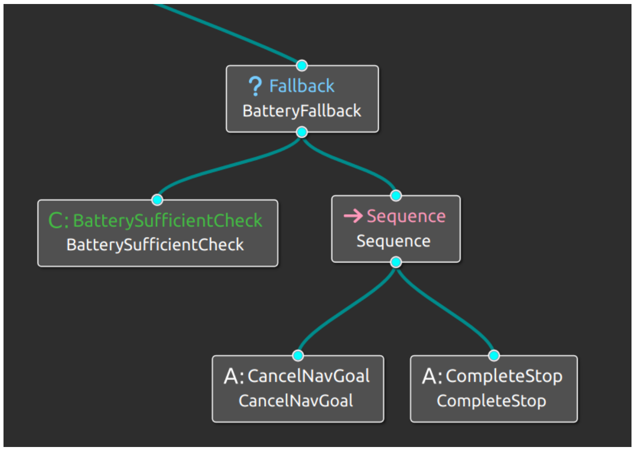


Figure 4.4: Battery Fallback Behavior

resulting function can be described as:

Formel here

On a real robot the average values for the idle and driving consumption could be experimentally determined. For the simulated turtlebot we created a simulated battery package which updates the battery charge every second. The battery package subscribes to the comman velocity topic and decreases the battery charge for every message that is received. We created the battery package because the behavior for this scenario needs to be tested. This is way the consumption rates are not close to reality. The battery package provides a service to get the battery charge and the values for the angenommenen consumption rates. 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, we implement a way to make the global planner more flexible. The execution checker component monitors the health of the Nav2 global planner. Additionally, the BT analyses the global planners output. If one of the condition fails, the given navigation goal is not reachable by the planning algorithm. We combat this by shifting the goal to a nearby alternative to get the robot as close to the original goal as possible. Is the location still unreachable, the goal gets moved further away and closer to the robot again. This routine gets repeated for up to ten times, which leads to a 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.

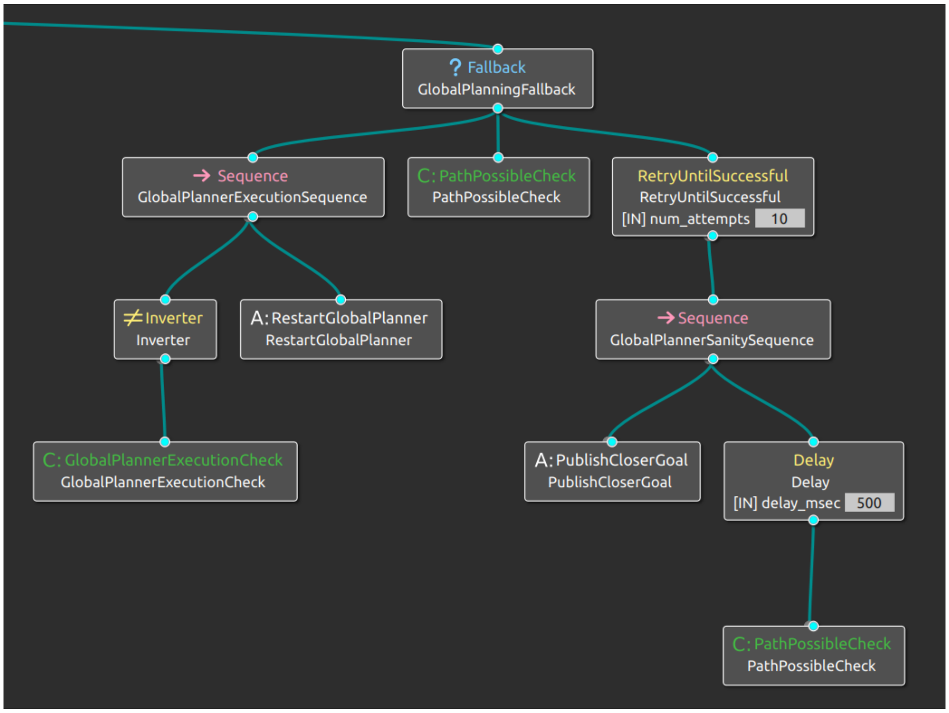

? Fallback
GlobalPlanningFallback
→ Sequence
GlobalPlannerExecutionSequence
C: PathPossibleCheck
PathPossibleCheck
RetryUntilSuccessful
RetryUntilSuccessful
[IN] num_attempts 10
≠ Inverter
Inverter
A: RestartGlobalPlanner
RestartGlobalPlanner
→ Sequence
GlobalPlannerSanitySequence
C: GlobalPlannerExecutionCheck
GlobalPlannerExecutionCheck
A: PublishCloserGoal
PublishCloserGoal
Delay
Delay
[IN] delay_msec 500
C: PathPossibleCheck
PathPossibleCheck

Figure 4.5: Path Planning Fallback

Chapter 5

Evaluation

5.1 Scenarios

Chapter 6

Summary

6.1 Conclusion

6.2 Prospect

Bibliography

[1] SAEInternational, *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, apr 2021.

[2] 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.

[3] 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)*, pp. 315–321, 2020.

[4] R. Arkin, R. Arkin, and R. Arkin, *Behavior-based Robotics.* Bradford book, MIT Press, 1998.

[5] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, *Modeling Software with Finite State Machines: A Practical Approach.* 05 2006.

[6] 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.

[7] 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.

[8] E. F. und Innovation (EFI), "Gutachten zu forschung, innovation und technologischer leistungsfaehigkeit deutschlands 2018," tech. rep., EFI, Berlin, 2018.

[9] R. Murphy, R. Murphy, and R. Arkin, *Introduction to AI Robotics.* A Bradford book, MIT Press, 2000.

[10] 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*, pp. 1–6, IEEE, 2020.

[11] 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*, pp. 36–43, 2008.

[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), pp. 625–635, Springer International Publishing, 2014.

[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*, pp. 1–8, 2017.

[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. 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, p. 5, Kobe, Japan, 2009.

[18] 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.