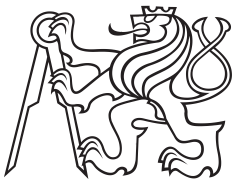
Bachelor Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

# Autonomous road crossing with a mobile robot

Jan Vlk

Supervisor: Mgr. Martin Pecka, Ph.D.
Field of study: Cybernetics and Robotics
May 2023

# BACHELOR‘S THESIS ASSIGNMENT

## I. Personal and study details

Student's name: **Vlk Jan** Personal ID number: **499227**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

## II. Bachelor’s thesis details

Bachelor’s thesis title in English:

**Autonomous Road Crossing with a Mobile Robot**

Bachelor’s thesis title in Czech:

**Autonomní přejezd silnice mobilním robotem**

Guidelines:

The goal of the thesis is design, implementation and experimental verification of an algorithm for safe crossing of roads with public traffic with a mid-sized mobile robot. The student should review existing approaches to the problem and assess their suitability for use on target robotic platforms. Also, a method for evaluating performance of road-crossing algorithms should be proposed and applied. In the implementation, sensor data from color cameras, 3D lidars and public cartographic data can be used to improve performance of the algorithm. The algorithm should also expect an input with poses and velocities of detected vehicles, although the detection itself is not a part of this thesis. Other contextual inputs can be given, like maximal or expected velocity of incoming vehicles, road type, number of lanes on the road or presence of a pedestrian crossing with or without traffic lights. Given this context and vehicle velocity data, the algorithm should be able to safely assess the situation and decide whether it is safe to cross the road in a given moment or not. In the safe case, a control algorithm should be developed that will perform the actual road crossing (with continuous checking of safety of the maneuver).
Experimental verification of the work should be done both in simulation and in a controlled real-world experiment. In the real-world experiment, the robot will not enter a real public driving road, but an experimental setup in a non-public area will be set up (in cooperation with thesis supervisor) to demonstrate behavior of the algorithm even in case of incoming traffic (which will be driven by faculty staff). Results of these experiments should be evaluated according to the proposed performance metric.

Bibliography / sources:

[1] https://wiki.openstreetmap.org
[2] J. Choi et al., "Environment-Detection-and-Mapping Algorithm for Autonomous Driving in Rural or Off-Road Environment," in IEEE Transactions on Intelligent Transportation Systems, vol. 13, no. 2, pp. 974-982, June 2012, DOI: 10.1109/TITS.2011.2179802.
[3] A. Chand and S. Yuta, "Navigation strategy and path planning for autonomous road crossing by outdoor mobile robots," 2011 15th International Conference on Advanced Robotics (ICAR), 2011, pp. 161-167, DOI: 10.1109/ICAR.2011.6088588.
[4] N. Radwan, W. Winterhalter, C. Dornhege and W. Burgard, "Why did the robot cross the road? —Learning from multi-modal sensor data for autonomous road crossing," 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2017, pp. 4737-4742, DOI:10.1109/IROS.2017.8206347.
[5] M. Colledanchise and P. Ögren Behavior Trees in Robotics and AI: An introduction, 2018, CRC Press, ISBN 9781138593732.

Name and workplace of bachelor's thesis supervisor:

**Mgr. Martin Pecka, Ph.D. Vision for Robotics and Autonomous Systems FEE**

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **03.02.2023** Deadline for bachelor thesis submission: _____

Assignment valid until: **22.09.2024**

| Mgr. Martin Pecka, Ph.D. | prof. Ing. Tomáš Svoboda, Ph.D. | prof. Mgr. Petr Páta, Ph.D. |
|---|---|---|
| Supervisor's signature | Head of department's signature | Dean's signature |

## III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

| . | |
|---|---|
| Date of assignment receipt | Student's signature |

# Acknowledgements

I want to express my sincere gratitude to everyone who supported me throughout the completion of this bachelor's thesis. My most profound appreciation goes to my advisor, Martin Pecka, for his invaluable guidance, feedback, and support throughout the research and development process.

I am also grateful to my family and friends for their unwavering support and motivation. Their love and encouragement kept me motivated to finish this thesis and achieve this milestone.

Finally, I want to acknowledge the faculty staff who generously shared their time for this research in its experimental phase. Their willingness to participate enabled me to conduct this study and make meaningful contributions to the field.

Thank you all for your support and encouragement!


# Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date May 9, 2023

_____

Signature

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 9. května 2023

_____

podpis autora práce

# Abstract

In this thesis, our task was to design, implement and evaluate an algorithm for the safe crossing of public roads with a middle-size mobile robot.
The first part of this task is to conclude whether it is safe to cross the road in the robot's current location. If it is, the second part of the task is developing a control algorithm to perform the movement needed to cross the road. Continuous monitoring of the traffic situation is necessary for the safety of the maneuver. We are also to provide evaluation metrics for determining the functionality and optimality of developed algorithms. The verification and evaluation of the developed algorithm will be conducted in simulation and a controlled, real-world experiment.



**Keywords:** Autonomous robot operation, behavior trees, collision avoidance, collision detection, finite-state machines, road crossing, ROS



**Supervisor:** Mgr. Martin Pecka, Ph.D.
Vision for Robotics and Autonomous Systems,
Resslova 9,
12000 Praha 2



# Abstrakt

V této práci je naším úkolem navrhnout, implementovat a vyhodnotit algoritmus pro bezpečný přejezd silnic s mobilním robotem střední velikosti.
První část úkolu je zjistit, zda je bezpečné přejít silnici v místě, kde se robot právě nachází. Pokud ano, druhá část úkolu je navrhnout algoritmus pro řízení pohybu potřebného k přejetí silnice. Pro bezpečnost manévru je nutné provádět kontinuální monitorování dopravní situace.
V neposlední řadě je třeba navrhnout metriku pro vyhodnocení funkčnosti a optimálnosti vyvinutých algoritmů. Verifikaci a vyhodnocení vyvinutého algoritmu provedeme nejprve v simulaci a následně i v kontrolovaném experimentu v reálném světě.



**Klíčová slova:** Autonomní operace robota, detekce kolizí, konečné stavové automaty, přejíždění silnic, ROS, rozhodovací stromy, vyhýbání se kolizím

**Překlad názvu:** Autonomní přejezd silnice mobilním robotem

# Contents

# Figures



# Tables

# Introduction

In today's world, mobile robots are increasingly being utilized in a variety of applications. In many of these applications, the robots must cross roads to achieve their goals, making it essential to design an algorithm that enables the robot to cross the road safely.
The algorithm should be able to determine whether the current place is suitable for crossing. The algorithm will accept contextual inputs such as vehicle velocity data, road type, number of lanes, and the presence of a pedestrian crossing with or without traffic lights. These data will be used to assess the current situation and determine whether it is safe to cross the road. If it is, it should facilitate the crossing itself. If the location is not suitable, the algorithm should provide a reason and suggest a more appropriate location nearby. The algorithm must also be designed to operate on different robots with various sensor configurations and road types without limitations.
This thesis aims to provide a theoretical background to the problem and explore possible solutions. We will also present the hardware and software used for real-world and simulation experiments. We will discuss our chosen approach and its functionality and present the algorithms we developed and implemented. Finally, we will explain the results of our experiments and discuss their significance.
Our work also depends on the output of other projects, such as vehicle detection and localization or path planning. We cannot rely on these projects to be completed or entirely functional. Therefore, we need a way to simulate and test our algorithm without them.
In simulation experiments, we will inject data directly into our algorithm. For real-world experiments, we will try to use the outcomes of the projects mentioned earlier. However, we can inject data directly into our algorithm, provided the projects are not finished or functional.

## Used abbreviations

- **AI** – Artificial Intelligence
- **API** – Application Programming Interface
- **BT** – Behavior Tree
- **CRAS** – Center for Robotics and Autonomous Systems
- **ENU** – East-North-Up
- **FSM** – Finite State Machine
- **GPS** – Global Positioning System
- **GUI** – Graphical User Interface
- **HFMS** – Hierarchical Finite State Machine
- **IMU** – Inertial Measurement Unit
- **LiDAR** – Laser imagining Detection and Ranging
- **NED** – North-East-Down
- **NPC** – Non-Player Character
- **OSM** – Open Street Map
- **REP** – ROS Enhancement Proposals
- **RL** – Reinforcement Learning
- **ROS** – Robot Operating System
- **TPI** – Terrain Profile Index
- **UTM** – Universal Transverse Mercator
- **WGS84** – World Geodetic System 1984
- **ZABAGED** – Základní báze geografických dat (Basic database of geographic data)

# Chapter 1

# Theoretical background

## 1.1 Behavior trees

A behavior tree (BT) is a way to structure algorithms – the switching between individual tasks in an autonomous agent. It was created to express behavior patterns for NPCs (non-player characters) in computer games. Since then, it has found many more applications, and nowadays, it is also widely used in robotics and AI applications.
BTs, as the name suggests, are tree-like structures where each node represents an action, a condition, a control, or a decorator node. Action and control nodes are leaves of the tree structure. Control nodes are used to control and modify the flow of the tree. Examples of these nodes are `sequence`, `fallback`, or `repeat`. Decorator nodes are used to modify the return values, thus modifying the behavior of its children. Examples of these nodes are `force-success`, `force-failure`, or `inverter`.
The execution of a BT commences at the root node and then progressively traverses the tree structure in a depth-first fashion polling its nodes. The nodes' polling, more frequently called the ticking, is periodically repeated. Each node, once ticked, begins its execution process, and once finished, it returns a status. This status can be either `SUCCESS`, `FAILURE`, or `RUNNING`. The action and control nodes are responsible for determining and returning these states. The control nodes alter the tree's flow and tick handling based on its children's return states. Decorator nodes modify the return states of their children. The return states of some nodes are shown in table 1.1.

| Node type | `SUCCESS` | `FAILURE` | `RUNNING` |
|---|---|---|---|
| `Action` | Action succeeds | Unable to complete | During completion |
| `Condition` | Condition is true | Condition is false | N/A |
| `Sequence` | All children succeed | One child fails | One child running |
| `Fallback` | One child succeeds | All children fail | One child running |
| `Parallel` | $N$ children succeed | $< N$ children succeed | Children running |
| `Repeat` | Child succeeds | Child fails $x$ times | Child running |

**Table 1.1:** Return states of some nodes.

The first chapter in [1] provides a more thorough explanation of the behavior trees.

### 1.1.1 Commonly used nodes

Here we will present the most commonly used nodes and their functionality.

**Sequence** – Control node that polls its children one at a time in a predefined order. If one of the children were to return `FAILURE`, the polling of other children is stopped, and the `sequence` node returns `FAILURE`. The same happens if one of the children returns `RUNNING`. If all children return `SUCCESS` the `sequence` node returns `SUCCESS`.

**Fallback** – Also known as `Selector` is a control node that polls its children one at a time in a predefined order. If one of the children were to return `SUCCESS`, the polling of other children is stopped, and the `fallback` node returns `SUCCESS`. The same happens if one of the children returns `RUNNING`. If all children return `FAILURE` the `fallback` node returns `FAILURE`.

**Parallel** – Control node that allows multiple actions to run concurrently. It returns `SUCCESS` if $N$ or more children return `SUCCESS` and `FAILURE` if less than $N$ children return `SUCCESS`. If all children return `RUNNING` the `parallel` node returns `RUNNING`.

**Repeat** – Control node that polls its child a specified number of times or until the child returns `SUCCESS`, whichever comes first. If the child returns `RUNNING`, the `repeat` node returns `RUNNING`. If the child does not return `SUCCESS` before the number of repetition is reached the `repeat` node returns `FAILURE`.

**Inverter** – Decorator node that inverts the return state of its child. If the child returns `SUCCESS`, the `inverter` node returns `FAILURE` and vice versa. If the child returns `RUNNING` the `inverter` node returns `RUNNING`.

**Force-success** – Decorator node that returns `SUCCESS` regardless of the return state of its child.

**Force-failure** – Decorator node that returns `FAILURE` regardless of the return state of its child.

### 1.1.2 Graphical representation of BTs

We will represent the BTs in this work in the following way. Action nodes will be rectangular with the name of the action written inside. Control nodes will be elliptical with the name of the condition written inside. Control and decorator nodes will be rectangular with a corresponding symbol inside. The symbols are shown in a table 1.2.

If the BT has a sub-tree in its structure, we will represent it as a diamond-shaped node with the sub-tree name written inside.

| Node type | Description | Symbol |
|---|---|---|
| `Root` | The root of the tree | *Root* |
| `Sequence` | Ticks its children if the return is `SUCCESS` | $\rightarrow$ |
| `SequenceStar` | Ticks its children if the return is `SUCCESS` | $\rightarrow^*$ |
| `Fallback` | Ticks its children if the return is `FAILURE` | ? |
| `Parallel` | Allows multiple actions to run concurently | $\rightrightarrows$ |
| `Repeat` | Repeats the child node $(x)$ times | $\circ(x)$ |
| `ForceSuccess` | Allways returns `SUCCESS` | $\checkmark$ |
| `ForceFailure` | Allways returns `FAILURE` | $\times$ |
| `Inverter` | Inverts the return value of its child | $\neq$ |

**Table 1.2:** Symbols used for control and decorator nodes in BTs.

### 1.1.3 BT example

We will present a simple example demonstrating the BTs structure and design principles.
The example BT is shown in figure 1.1. This BT was created in the algorithm design's beginning phase, and its modified version will be presented later as it is used in the final implementation. The goal of this sub-tree was to position the robot so that it would cross the road as fast as possible, meaning we want the robot to stand perpendicular to the road.

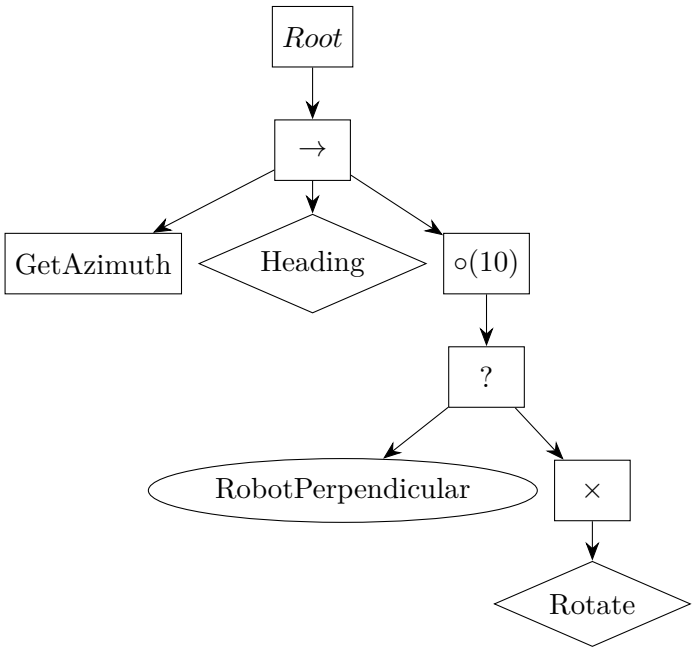



**Figure 1.1:** BT example.

We start in the root node and continue straight to the `sequence` node. From

there, we go to the action node GetAzimuth, which gives us the current heading of the robot. If the execution of the GetAzimuth node is successful, we continue to the Heading sub-tree node. This sub-tree aims to calculate the heading the robot needs to achieve in order to be perpendicular. If all nodes inside the sub-tree are successful, we continue to the `repeat` node. This node will repeat its children ten times (or less if success is achieved sooner). The first child we will tick is a `fallback` node, with its first child being a condition node RobotPerpendicular. The condition node return value states whether or not the robot is already perpendicular to the road we mean to cross. If we are not yet perpendicular, we continue. The next node we tick is a `force-failure` node with a sub-tree node as its child. The sub-tree is responsible for rotating the robot to the desired heading.

### 1.1.4 Other used BT nodes

Here we will present other BT nodes. These nodes are an expansion of the common ones and are implementation specific.

**SequenceStar** ($\rightarrow^*$) – Also know as `SequenceWithMemory`, a control node that functions in the same way as `Sequence`. The only difference is that this node does not repeat children that returned `SUCCESS` until all children have. Meaning until the `SequenceStar` node returns `SUCCESS`, it will tick only the children that have not succeeded yet.

**ReturnSuccess** – A leaf node that returns `SUCCESS` once ticked. We will represent this node as an ellipse with a checkmark character ($\checkmark$) inside.

**ReturnFailure** – A leaf node that returns `FAILURE` once ticked. We will represent this node as an ellipse with a cross character ($\times$) inside.

### 1.1.5 Common BT structures

Common programming principles can explain some BT structures. We will present a few of these structures that we have used in our BT structure.

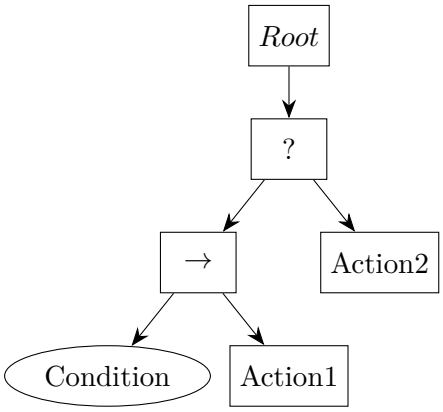
#### If-else

The `if-else` structure starts with a `Fallback` node, and the first child a `Sequence` node with its first child a condition node and second child an action node, ticked if the condition is true. The second child of the `Fallback` node is also an action node that is performed if the condition is false.
The structure is shown in figure 1.2a.

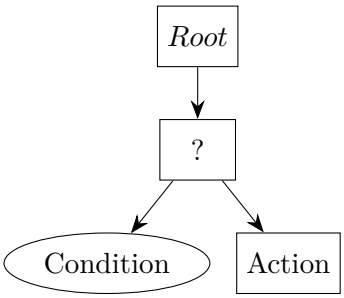
#### Condition-action

We could also name this structure as `if not`.
The `condition-action` structure starts with a `Fallback` node. Its first child is a condition node, and its second is an action node. The idea behind this structure is to check if an action has been performed, and if it has not, we

want to perform it.
The structure is shown in figure 1.2b.




**(a) :** The `if-else` structure.




**(b) :** The `condition-action` structure.

**Figure 1.2:** The common structures used in the creation of BTs.

## 1.2 Finite-state machines

Finite-state machines (FSM) are a mathematical model of computation. They are used to model the behavior of a system in a finite number of pre-defined states. The system can be in only one state at a time. In each state, a computation or an action is performed. The change of a state is possible only via predetermined transitions triggered by a condition.
FSMs are a common method of describing and solving high-level sequential control problems. They are used in many fields, such as robotics, computer science, electrical engineering, etc.
FSM offers a very effective method in the implementation of complex robot behavior in comparison to monolithic programming.[2] Moreover, the learning curve for using FSM is minor; it is quite likely the reader already knows about FSMs from math or logic courses. Secondly, the integration itself is almost painless, especially when one takes the FSM into account from the early stages of the design.[3]
However, the FSMs are unsuitable for large and complex systems as they tend to become unmanageable and difficult to extend and reuse. The unsuitability becomes more evident for a fully reactive system, where each state must be able to transition to any other state. Such a condition imposes the FSM to become a fully connected graph ($\mathcal{O}(n^2)$). Maintaining and modifying such a graph is quite a labor-intensive and error-prone task.
FSMs are also unsuitable for systems requiring a high degree of autonomy. The FSMs are not able to learn and adapt to the changing environment.
The formal definition of an FSM and several examples can be found in [3].

### 1.2.1 FSM example

Here we will show the FSM for the example BT (figure 1.1) from the previous section.

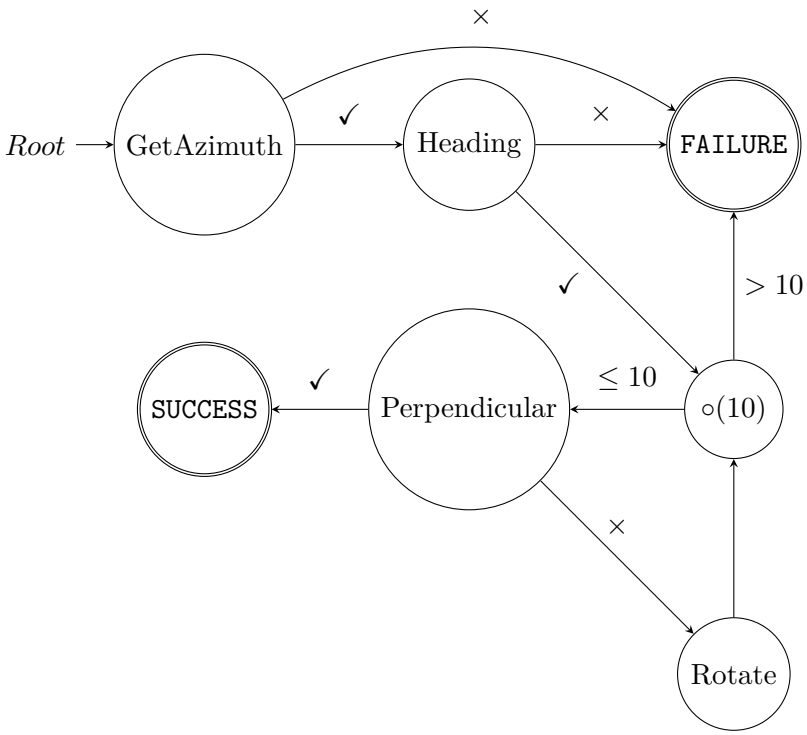



**Figure 1.3:** FSM for the example BT.

This is not a typical representation of an FSM. It is a one-to-one rewrite of the BT example. If we were to develop the algorithm using FSMs, the FSM would look different. BTs and FSMs require different mindsets and design principles, and while they may be transformed from one to the other, it usually results in a nonoptimal structure.

## 1.3 Hierarchical FSMs

Hierarchical state machines (HFSM), also known as statecharts, were developed to alleviate the cumbersome transition duplication required in large FSMs and add structure to aid comprehension of complex systems. It clusters states into a group (named superstate) where all the underlying internal states (substates) implicitly share the same superstate.[4]
While the HFSM solves the problem of transition duplication, it does not solve the complexity problem. The HFSM is still a fully connected graph unsuitable for large and complex systems.

## 1.4 Comparison and chosen approach

We can use several design approaches to solve the task of crossing a street. We will briefly present them and state a few advantages and disadvantages of each one. We will mainly use the informations and insights from [1].

### Monolithic approach

We can use a monolithic approach, where we write a single program that will handle all the tasks.
This approach is the most straightforward and easiest to implement but is not very flexible. It would be complicated to modify or extend the abilities of our program to the point where we would be forced to rewrite it in its entirety. This approach also generates a design that is not easily readable, and it would be almost impossible to find and correct bugs and glitches.
For all those reasons, the monolithic approach is unsuitable for anything other than elementary systems, and we will not use it for our solution.

### FSM approach

The second approach is to use an FSM. More specifically, HFSM as it is an improvement over FSM and addresses a few of the FSM issues.
The advantages of HFSMs are that the structure is intuitive and generally easy to understand. Being common in many parts of computer science and used for quite some time, they are also easy to implement. FSMs also offer good flexibility and maintainability for many problems.
The main disadvantage of HFSMs is that the flexibility is limited to certain areas of use and systems with limited scale. It is impossible to add new states or transitions in complex systems easily.
The scalability of FSMs is also a problem. With rising demands on agent AI complexity, game programmers found that the FSMs that they used scaled poorly and were difficult to extend, adapt and reuse.[5]
The FSM's poor scalability makes this approach unsuitable for our solution.

### BT approach

The third approach is to design the algorithm in the form of a BT.
The advantages of this approach are its modularity, reusability, reactivity, readability, and scalability. Modularity is closely linked with reusability. The design principles of BTs allow us to decompose the algorithm into sub-trees which may be implemented and tested separately. Decomposition also allows us to tackle large complex systems with relative ease. The BTs are reactive in the sense that they can react quickly and effectively to the changing environment. Even though they require a different design approach than FSMs, they provide a coherent and compact structure that is easy to understand and maintain.
The main disadvantage of BTs is that they are not very common in the industry and are not as well known as FSMs. For this reason, the tools and libraries are not as numerous or mature as those available for FSMs.

As mentioned earlier, they are different from FSMs and, as such, require a different approach to designing an optimal solution.

### Chosen approach

The approach we have chosen to use in this thesis is the BT approach. We have chosen it for many reasons, mainly its scalability, readability, and maintainability of large complex systems. The BTs are also very flexible and can be easily extended and modified. The supervisor also suggested the use of the BT approach.

## 1.5 Mathematical aparatus

This thesis will use chapters from linear algebra, calculus, and geometry. The used theory will be shown and briefly explained in sections where it is needed. We will not provide precise definitions, or state used theorems as it is outside the scope of this work. However, a book, a paper, or an online document with the corresponding information will be provided should the reader require a more thorough explanation.

## 1.6 Maps

We will use the maps from the OpenStreetMap (OSM) project[1]. The maps will be used to determine the surroundings of the robot and whether the current position is suitable for crossing.
OSM is a project that creates and distributes free geographic data. The data is created by the community of users and is available for anyone to use. [6]
The map data are expressed by a node, a way, or a relation. A node is a singular point in a map, it could be a landmark, a corner of a building, or a spot on the road. A way is an object created from multiple nodes. It can be either closed or open. Closed ways may represent a park, building, or other type of area. Open ways commonly represent roads, rivers, or other linear features. The relation is a collection of nodes, ways, or other relations. It is used to describe more complex objects, such as a bus line, a building complex, etc.
We must also clarify the terminology we will use regarding azimuth and heading. We will use the definitions stated here [7].
An azimuth is a bearing, more precisely, a compass bearing from a specific point of observation like a radar station. A heading (in the general case of moving "forward") is the direction your nose is pointed in.

[1] `https://www.openstreetmap.org`

# Chapter 2

# Used hardware and software

## 2.1 Software

All programming work in this thesis is aimed to work with the Robot Operating System (ROS) [8]. More specifically, we will use ROS1 in version Noetic Ninjemys[1].

### Programming languages

The majority of implementation work will be done in a C++ programming language. The version of C++ standard used is C++14, as it is the default for the ROS version we use.
The C++ language was chosen for its speed and efficiency. It was also chosen for compatibility with some of the libraries we need to use for our project.
The second programming language we will use is Python in version 3.8. Python was chosen for its simplicity and ease of use, as well as for integrating our previous work in OSM data processing.

### BT library

There are a few possibilities regarding the BT library we can use for our solution. As BTs are not very commonly used, the choice is more limited than if we were to use an FSM. Another limiting factor we imposed is support or direct integration with ROS.
We still have a few options, and we can even choose a programming language in which to implement the BT nodes. The two programming languages with the most library options are C++ and Python. This copies the ROS mentality, where these two languages are natively supported. Several available BT libraries are discussed here [9].
We have decided to use a C++ behaviortree-cpp-v3 library[2]. The choice was made for multiple reasons. This library was written with deployment in ROS in mind. Moreover, it is regularly updated and maintained, making it a safe choice for us. It also comes with documentation that will be helpful during the implementation process. There are two versions of the documentation

[1] `http://wiki.ros.org/noetic`
[2] `https://github.com/BehaviorTree/BehaviorTree.CPP`

[10] and [11]. We will mainly use the newer one (the second mentioned), but we will cross-reference it with the older one.
Another benefit of this implementation is that it comes with a GUI application for creating BTs called Groot[3]. This application creates an `.xml` file with the BT structure we can import into our code later.

#### Libraries for OSM and work with geographical data

There are a lot of libraries to choose from when it comes to working with OSM data. These libraries are created for different programming languages and have different features.
Even though the majority of our work was written in C++, we were building on top of previous work of assigning costs to road segments in OSM data. This work was done during the 2022 summer as a part of the RobInGas project here at the CTU under the Center for Robotics and Autonomous Systems (CRAS[4]) group.
The work was done in Python, and the library used was the overpy library[5]. This library is used to access the OSM Overpass API and download the map data. The Overpass API (formerly known as OSM Server Side Scripting) is a read-only API that serves up custom-selected parts of the OSM map data. The difference between the main API is that the Overpass API is optimized for small to large consumers (up to roughly 10 million elements). Many services and applications use it as a database backend.[12]
Other libraries used for work with the OSM data were shapely[13] and numpy[14]. These libraries were used to classify and assign costs to individual road segments in the downloaded OSM data.
In our work, we also need to convert the coordinates of the robot from the GPS coordinate system to the UTM coordinate system. The conversion is done using the GeographicLib library[15] in C++ and utm[6] in Python.

## 2.2 Hardware for real-world experiments

### 2.2.1 Robots

In this section, we will present the robots available for use in the real-world experiments. We have two robotic platforms available for us: Husky and Spot.

#### Husky

Husky is a medium all-terrain robot developed by Clearpath Robotics. It is a four-wheeled robot with a payload capacity of 75 kg. The weight of this robot without the payload is 50 kg, and its maximal speed is 1 [m/s]. This robot is mainly used outside of urban areas. The photo of the Husky in the configuration we use is shown in figure 2.1a.

[3] `https://github.com/BehaviorTree/Groot`
[4] `https://robotics.fel.cvut.cz/cras`
[5] `https://github.com/DinoTools/python-overpy`
[6] `https://github.com/Turbo87/utm`

More information about the Husky platform is available at the Clearpath Robotics website[7].

### Spot

The Spot is a medium all-terrain robot developed by Boston Dynamics. It is a four-legged robot with a payload capacity of 14 kg. The weight of this robot without the payload is 33 kg, and its maximal speed is 1.6 [m/s]. This robot is designed to be mainly used in urban and industrial areas. The photo of the Spot robot with our payload is shown in figure 2.1b.
More information is available at the Boston Dynamics website[8].

**(a) :** The Husky robot configuration. **(b) :** The Spot robot configuration.

**Figure 2.1:** Robots used in the real-world experiments.

The photos are courtesy of CRAS at FEE CTU.

## 2.2.2 Sensors

The capabilities of our robots are highly dependent on the sensors we attach to them. Without them, the possibilities and options for missions are minimal. In our work, we will use some sensors directly and some indirectly. The indirect usage of sensors is connected with dependencies on other projects. One notable example is the detection of vehicles and other obstacles. This detection is not in the scope of our work but is instrumental to its success.

### Magnetometer

This is one of the sensors we use directly. We use it to determine the azimuth of our robot and help it position itself perpendicular to the road it will try to cross.
We say that the sensor is used directly. However, the transformation of the IMU magnetometer data into the azimuth was not implemented as a part of this work.

[7] https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/
[8] https://www.bostondynamics.com/sites/default/files/inline-files/spot-specifications.pdf

### Camera

Our robots are fitted with cameras pointing forward, backward, left, right, and up. This sensor is mostly used to determine the classification of obstacles rather than detecting the obstacles themselves. As this sensor is not vital to the functionality of our algorithm, we will not discuss them further.
The cameras on our robots are GigE Basler ace2 PRO.

### LiDAR

Another sensor our robots are equipped with is LiDAR. This sensor detects incoming vehicles and determines their speed and position vectors and other parameters. It is probably the most important sensor for our algorithm, as its processed data are being used as our algorithm's main switching condition.
The function and scanning mechanisms of LiDARs are described in [16].
The LiDARs used on our robots are Ouster OS0-128.

### GPS

All robots also have a GPS sensor. We use this sensor for precise localization of the robots in the global coordinate system.
The GPS sensors we use are Emlid Reach M+.

## 2.3 Simulation environment

We will simulate the behavior of our robot in the Gazebo simulator[9]. Gazebo is a 3D simulator for robots. Its biggest advantage is its direct integration with ROS. This means that we can simulate similar behavior to the one expected of the robot in real-world experiments.
We will use the Husky robot model for our simulations. The Husky was chosen as it is one of the robots we may use in the following real-world experiments, and its model was available to us.
As the creation and implementation of the simulation were not the main focus of this thesis, we have used previously created simulation environments. As the basis for our simulations, we used the `robingas_mission_gazebo` project[10]. This project was created by the CTU CRAS group. We have modified the project to fit our needs.
The simulation project was named `road_crossing_gazebo` and is available on GitHub[11].

[9] `https://gazebosim.org/`
[10] `https://github.com/ctu-vras/robingas_mission_gazebo`
[11] `https://github.com/vlk-jan/road_crossing_gazebo`

# Chapter 3

# Behavior tree algorithm structure

One of the most important parts of this thesis is the design of the BT algorithm. This chapter aims to present the design we created and provide the reasoning behind it.

## 3.1 Creating a behavior tree structure

First, we need to choose the correct approach to designing the BT structure. There are several possible approaches to creating a BT structure. We will discuss a few of these approaches and state the used one. The insight from [17], was instrumental for the selection and overview in this section.
The first approach is creating the complete BT structure by hand. Meaning we have to design every node, its position, and its function within the structure. This approach is the easiest but more time-consuming and error-prone than others.
The second approach is creating an initial BT and letting RL algorithms improve the BT's functionality and optimality. There are several options for this particular approach, as multiple possible RL algorithms exist for this task.
The third possible approach is constructing the BT from previously recorded human behavior. This approach also uses RL algorithms to transform the recorded behavior into a BT structure.
The last possible approach lets an RL algorithm construct the BT structure from the ground up.
Each of the presented approaches has its advantages and disadvantages. It is, therefore, vital to select the correct approach based on the possibilities and requirements of the task.

### Chosen approach

We have chosen the first approach, meaning we will construct the whole tree structure by hand. This was done as it is the easiest approach to this task and requires no additional steps.
Using different approaches to designing and improving the BT structure may be an interesting task for future work.
We will design the BT structure in the GUI application designed alongside

our chosen BT library, Groot. The application interface is shown in figure 3.1.

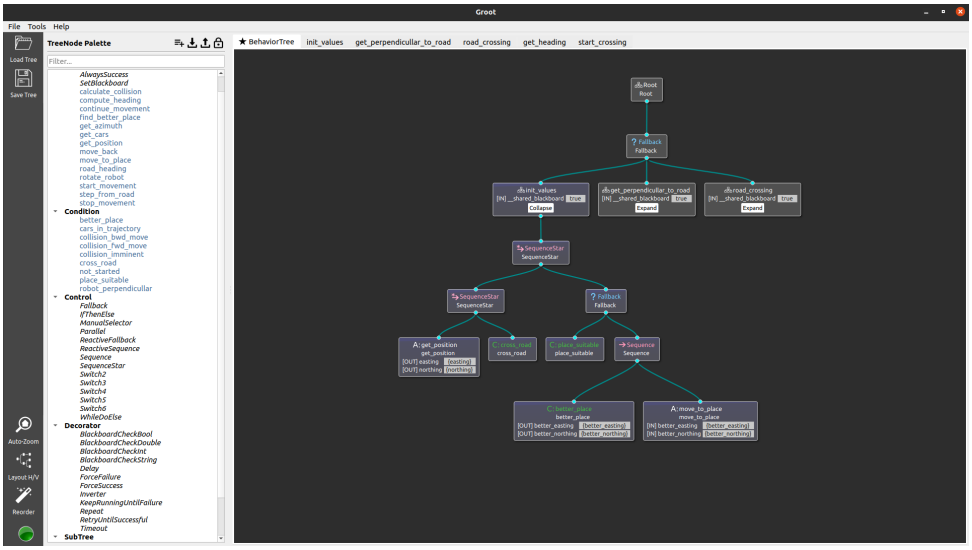



**Figure 3.1:** The Groot application interface.

## 3.2 Structure hierarchy – Main BT

We will divide the whole tree structure into several sub-trees to help with readability, modularity, and maintainability.
The first sub-tree will accomplish the initialization and will be responsible for determining whether the crossing should commence. It is also responsible for navigating the robot to a suitable crossing place. This sub-tree will be called `Init-BT`.
The second sub-tree is responsible for positioning the robot such that it is perpendicular to the road it is trying to cross. This sub-tree will be called `Perpendicular-BT`.
The third sub-tree is responsible for the navigation of the robot during the crossing. It will check the positions and speeds of incoming traffic and determine the best strategy for the crossing. This sub-tree will be called `Crossing-BT`.
There are a few more sub-trees in our structure, but as those are not the main ones, we will not present them here. They will be presented when they are mentioned in the main sub-trees structure. Their main task is to help with the modularity and reusability of the behavior they encode.
The main BT is shown in figure 3.2.
The main BT starts with a `Sequence` node. First, we need to check if the algorithm should be even started – to avoid collision between two nodes trying to control the robot. This we achieve with a condition node `StartAlgorithm`.

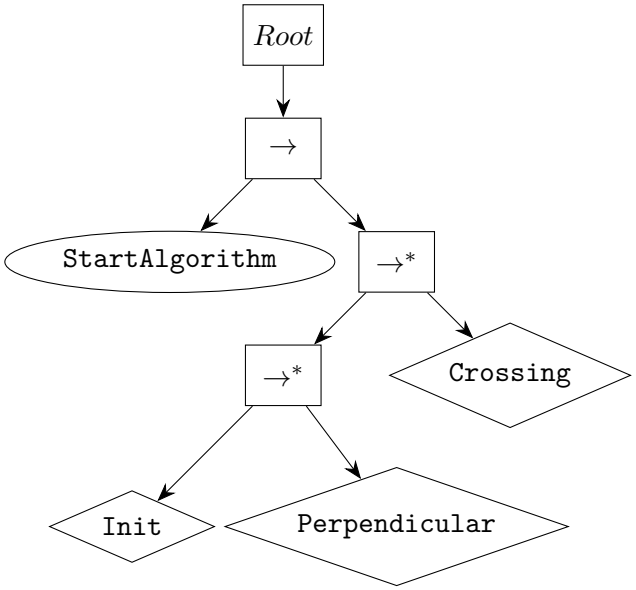



**Figure 3.2:** Main BT structure.

This node will check if the algorithm should be started. If it should not, the algorithm will not progress. The second child is a `SequenceStar` node. This node will tick the sub-trees responsible for the whole algorithm.
However, the first child is again a `SequenceStar` node. This is done to ensure that each preparation sub-trees will be successfully executed only once (if they return `SUCCESS`).
The first preparation sub-tree is the `Init-BT`, its structure shown in chapter 3.3 and its implementation in chapter 4.1.3.
The second preparation sub-tree is the `Perpendicular-BT`, its structure shown in chapter 3.4 and its implementation in chapter 4.1.4.
The last sub-tree is the `Crossing-BT`, its structure shown in chapter 3.5 and its implementation in chapter 4.1.5.

## 3.3 Init BT

As mentioned earlier, this BT is responsible for determining if we should start the crossing and for navigating the robot to the optimal location. This tree will be executed only once for each crossing. The `Init-BT` structure is shown in figure 3.3.
The polling of nodes in the structure is done in the following way. We start at a `Sequence` node. With its first child, a `SequenceStar` control node, we start the `Init-BT`'s first branch. The first node in this branch is an action node `GetPosition` followed by a condition node `CrossRoad`. The idea behind this branch is to determine the proximity of the robot to the road. If the robot is too far away from the road, the algorithm should not progress. This will help combat the possibility of trying to cross the wrong road, should it happen that two roads are close by.

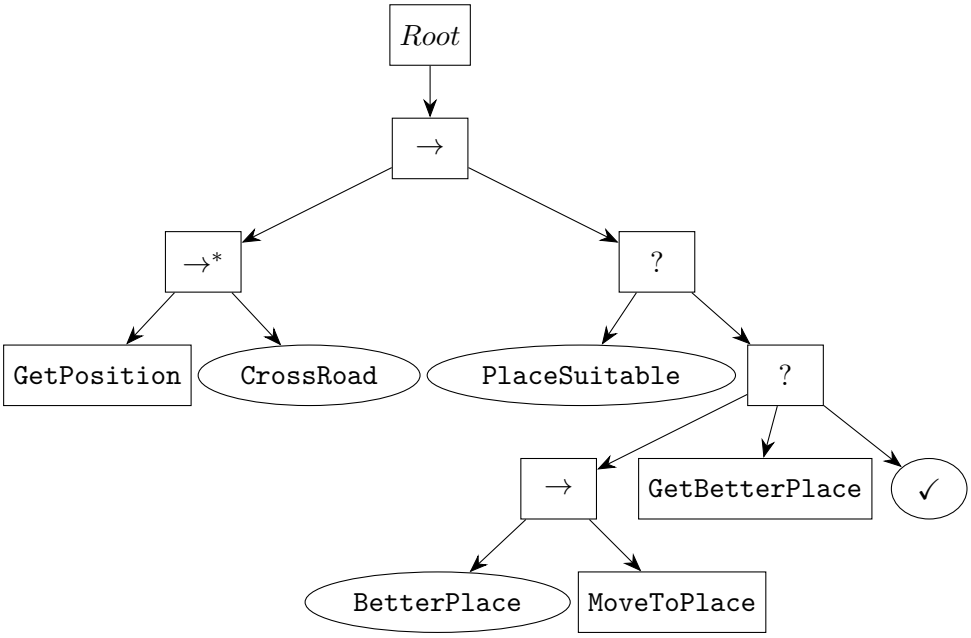



**Figure 3.3:** The Init-BT structure.

The second branch of this sub-tree starts with a `Fallback` node. The goal of this branch is to place the robot in an ideal position for crossing. This action should have been done before the mission, and the robot should have been sent to the optimal location by a path-planning node.
However, if such pre-mission planning was not performed, the `PlaceSuitable` condition node will check if the place is suitable. If not, the `BetterPlace` condition node will return if a better location has already been found. An action node `MoveToPlace` will steer the robot to a better location if it has been found. If it has not, the action node `GetBetterPlace` will try to find a more suitable place.
We will not perform the sub-tree again if a better place cannot be located. Instead, we will move on to the following sub-tree and cross the road in the position the robot is currently situated. This is done to avoid an infinite loop and is achieved with a `ReturnSuccess` node at the end of the second branch.

## 3.4 Perpendicular BT

This sub-tree is responsible for positioning the robot in the most optimal way for crossing the road. We have determined that to be the one in which the robot will cross the road the fastest. As such, the robot's heading should be perpendicular to the road it will cross. Figure 3.4 shows the BT structure for achieving so.
The first branch of this tree only needs to be ticked once in each run of the crossing algorithm. Therefore, we have a `SequenceStar` control node after the `Root` node.
The primary responsibility of the first branch is to calculate the azimuth

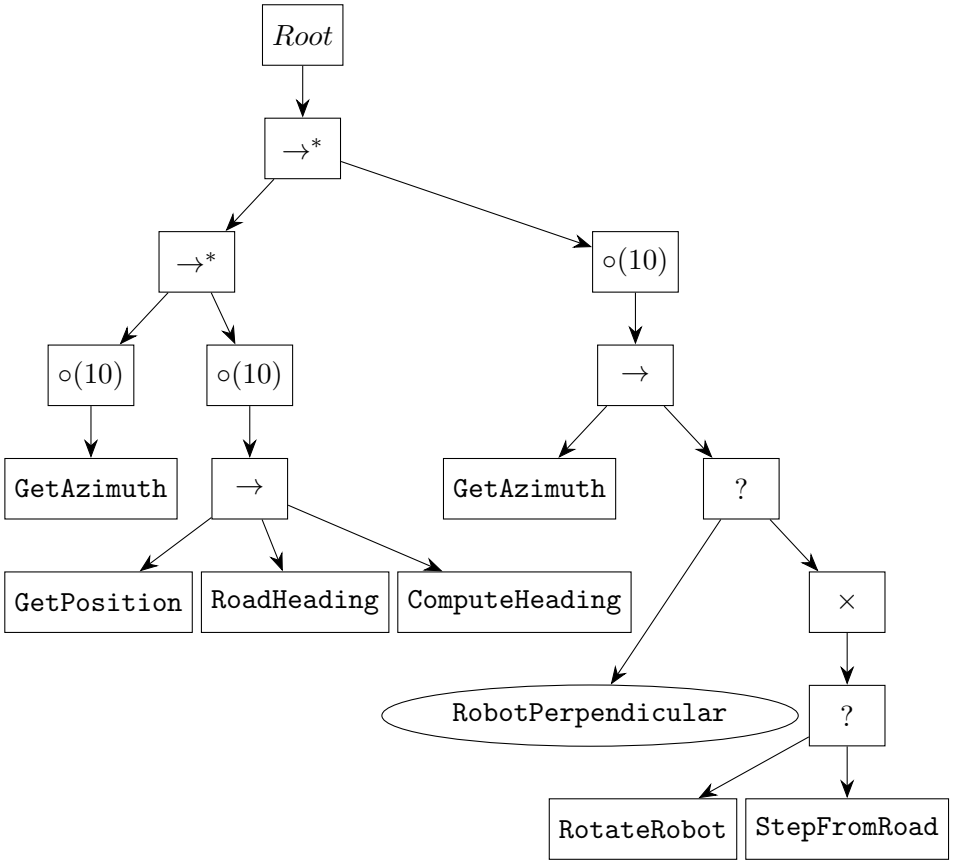



**Figure 3.4:** The Perpendicular-BT structure.

that will position the robot perpendicular to the road. As obtaining the robot's azimuth does not need to be repeated once successful, we start the branch with a `SequenceStar` node. The node has two children, both being a `Repeat` control node. The action to be repeated behind the first node is the obtaining of the robot's azimuth. The actions behind the second `Repeat` node are connected in a `Sequence`. This part of the algorithm calculates the optimal azimuth for the robot. Firstly we need to obtain the robot's position – `GetPosition` action node. Next, we need to determine the heading of the road closest to the robot. For this purpose, we have the action node `RoadHeading`. Finally, we calculate the optimal heading for the robot with the action node `ComputeHeading`. This concludes the left branch of our Perpendicular-BT.

The right branch starts with a `Repeat` node, followed by a `Sequence` node. The idea behind this branch is to utilize the heading value computed in the left branch and orient the robot accordingly. Firstly we need to obtain the robot's azimuth with the `GetAzimuth` action node. While this might seem redundant, we have just got the azimuth for calculation, it is vital to update the current azimuth as the value of obtained azimuth is only valid in the first run of the second branch. After receiving the current azimuth, we follow with a `Fallback` node and its first child, a condition node `RobotPerpendicular`.

This node tells us if the robot has achieved the optimal heading we calculated earlier. If not, we continue to the last part of this sub-tree.
The last part shall always return `FAILURE` and is responsible for the movement of the robot. This is necessary because we check the correct position before the movement. Firstly we try to rotate the robot with an action node `RotateRobot`. If the rotation was unsuccessful, we try to move the robot away from the road with an action `StepFromRoad`. The last action is a precaution, as the robot's rotation might have moved the robot onto the road, which is forbidden.
While we could unite the first two `Repeat` nodes, maybe even all three, we chose not to do so. The reason for not merging the nodes is to allow each part of the algorithm to fail independently. The number of repetitions for each node was set to 10, which we determined to be the optimal value.

## 3.5 Crossing BT

This tree is the most important part of the whole algorithm as it facilitates the road crossing. Figure 3.5 shows the structure of the sub-tree.
This tree starts with a `Sequence` node with three children. In the structure of this tree, there are two further sub-trees. These sub-trees are shown in figures 3.6a and 3.6b, and will be explained separately at the end of this section.
The first branch of this tree is responsible for obtaining the data of all detected vehicles from other ROS nodes. This functionality is implemented in just one action node `GetCars`.
The second branch starts with a `Fallback` node with two further sub-branches both behind `Sequence` nodes. This branch is responsible for the decision-making and control of the robot's movement.
The first sub-branch of the second branch deals with movement if no cars are present. First, we check this condition with a condition node `CarsInTrajectory`. If there are no vehicles, we are free to continue with our movement or start it if we have not done so yet. This is performed with a sub-tree `StartCrossing`, followed by the `MoveFwdFull` action node. This action node controls the robot to move forward with the highest velocity possible.
The second sub-branch is the decision-making part of the tree. First, it checks the movement status with a `StartCrossing` sub-tree. It follows with the `CalculateCollision` action node. The function of this node is to compute the velocities required for the robot to collide with all vehicles that have been detected. These velocities are vital information on which the decision-making is based.
The third child of the `Sequence` node is a structure of cascading if-else statements. The structure gradually checks the following conditions and performs the corresponding actions based on the results.
The first condition is the `CollisionFwdMove`. This node checks if the robot is about to collide with any of the detected vehicles in front. If it is not, the robot will continue moving forward.
If a collision is detected, we check if another collision would occur if we were

to stop the robot. A condition node `CollisionOnStop` is responsible for the detection. If not, we can stop the robot.
If there would be a collision if we were to stop, we would check if the robot could move backward instead. This is done with a condition node `CollisionBwdMove`. If the robot can move backward, we do so. If there would be a collision, we will stop the robot instead.

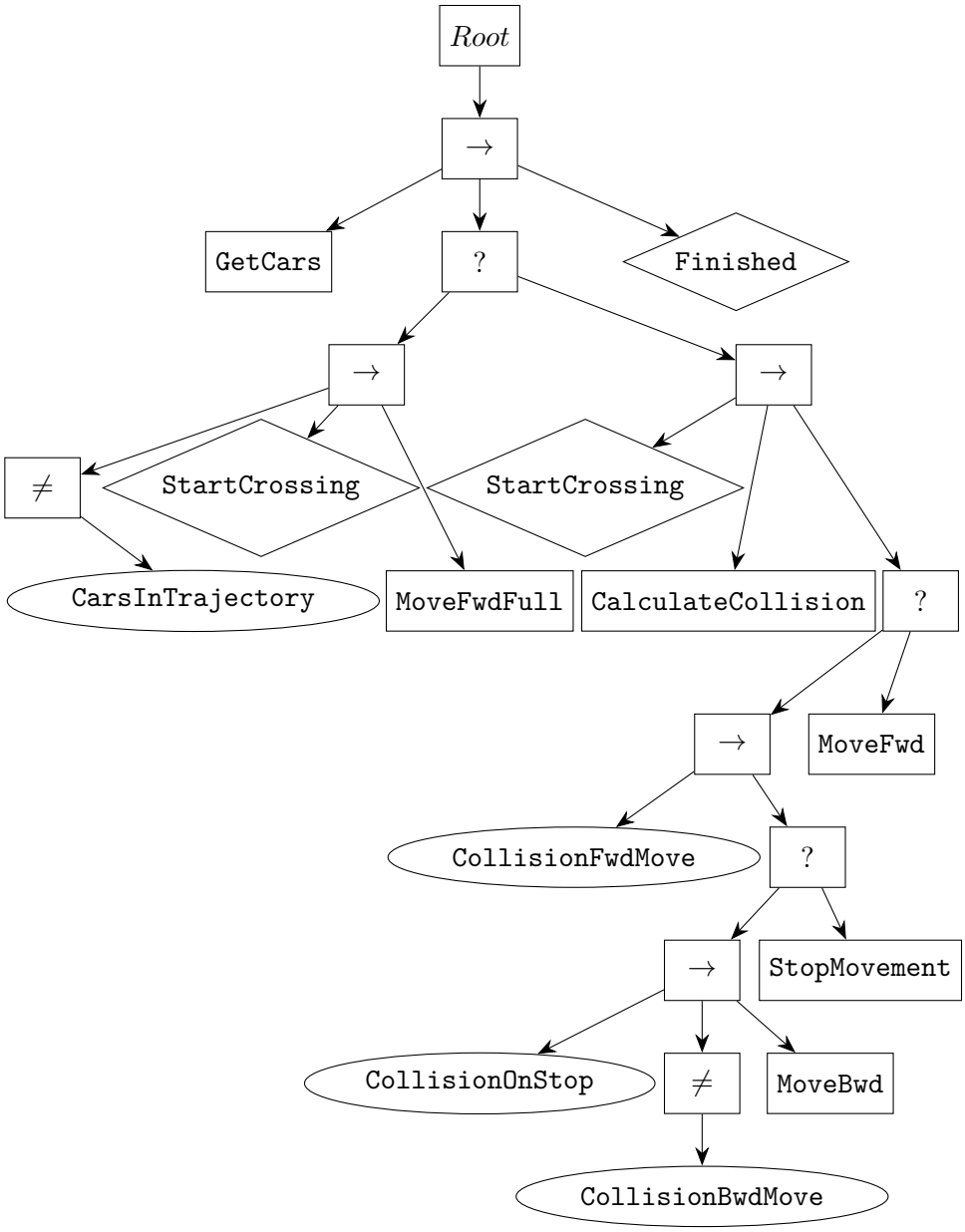
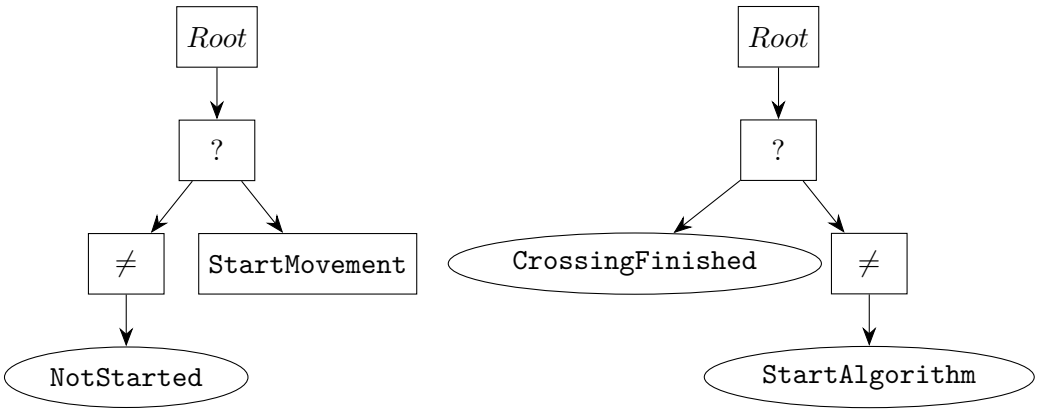



**Figure 3.5:** The Crossing-BT structure.

### 3.5.1 Crossing BT sub-trees




**(a) :** The StartMovement-BT structure. **(b) :** The Finished-BT structure.

**Figure 3.6:** The structures of sub-trees inside the Crossing-BT.

#### StartMovement BT

The `StartMovement` sub-tree is responsible for detecting if the movement process has started (the `NotStarted` condition node). If not, it starts the movement (the `StartMovement` action node).

#### Finished BT

The `Finished` sub-tree detects if the robot has crossed the road. There are two ways we can detect if the road was crossed. The first condition node `CrossingFinished` performs the check with the current GPS coordinates of the robot and road data, namely its GPS coordinates and width. The second check is the condition node `StartAlgorithm`, where the condition could be set from outside the algorithm, for example, from a different ROS node.

# Chapter 4

# Nodes implementation

This chapter will provide the implementation details for individual nodes used in our BT algorithm. We will split these descriptions into BT nodes and auxiliary functions used in the nodes. We will also provide the implementation details for ROS nodes and other ROS-related parts.

## 4.1 Behavior tree nodes

Here we will present the implementation of individual nodes in our algorithm. We will split the nodes into categories based on the sub-tree they belong to. We will also show the basic functionality of the library used.

### 4.1.1 Introduction

The BT algorithm is implemented using the behaviortree-cpp-v3 library. Therefore, we will present how we create, implement and start polling the tree. We will also show how nodes are created, implemented, and used.

#### Creating the node

To create the node, we need to create a class that inherits from one of the parent classes in the library. The parent classes depend on the type of node we want to create. If we create an action node, we inherit from the `SyncActionNode` class. If we create a condition node, we will inherit from the `ConditionNode` class.
There are two mandatory functions for all nodes, the `tick` and `providedPorts` functions. These two functions must be implemented, or an error will occur. The `tick` function is responsible for the actual implementation of the node. It is called every time the node is polled. It is also responsible for returning the state of the node.
The `providedPorts` function defines the ports the node will use. This function must be defined even if the node does not use any ports.

#### Using the node

To use a node, we must register it in the `BehaviorTreeFactory` object. We do this by calling the `registerNodeType` function. For this function, we need

to specify the class of the node we want to register, e.g., one of the nodes we created. It also takes one `std::string` parameter, which identifies the name of the node in our BT algorithm `.xml` file.

#### Creating and running the tree

To create the tree, we first must have a `.xml` file with the tree structure. This file is then parsed by the `BehaviorTreeFactory` object using the `createTreeFromFile` function. The result is a `Tree` object which we can store and later run. Ahead of calling the parsing for the tree file, all the nodes used in the file must be registered.
To run the tree, we call the `tickRoot` function on the `Tree` object. This function returns the state of the root node of the tree.

#### Blackboard

The blackboard is a shared memory between the nodes of the tree. It is used to store data that is being utilized by multiple nodes.
It is also the reason why we implement the `providedPorts` function. This function specifies the ports the node will use. The ports can serve as an `input`, an `output`, or both. `Input` ports can take a constant value (specified in the `.xml` file of the tree structure) or look for a value in the blackboard.

#### Logging

The BT library also provides us with logging functionality. This functionality is useful for debugging and testing.
We can use different types of loggers. The most common one is the `StdCoutLogger` which logs to the standard output. We can also use the `FileLogger`, which saves logs to a file.
We will use the `FileLogger` to log the tree execution. This logger is useful for its integration with the `Groot` application as we can import the produced file and visualize the polling of our BT structure.
The logging will be used only for debugging and testing purposes and serves no other function in the final product.

### 4.1.2 Main BT

In this BT, we only have one non-sub-tree node. This is because this sub-tree encodes the algorithm's structure rather than having nodes for execution.

#### StartAlgorithm – Condition node

This node is responsible for determining whether the robot is in the phase of crossing the road during its mission. It is necessary to implement such a node to facilitate the transfer of control from path planning to road crossing. In contrast, we could determine if the crossing should start based on the distance of the robot from the road. However, this method would fail whenever our robot has to walk alongside any road.
This node is implemented as a ROS service updating a static variable, which is checked when the node is ticked.

This service should be used mainly by other nodes outside of the package itself, with one notable exception. The exception being the very last node of our tree. It serves as a prevention against the looping of our algorithm.

### 4.1.3 Init BT

Here we will present the nodes used in the Init sub-tree. This sub-tree is used to initialize the BT algorithm. It is the first sub-tree to be executed. This tree is going to be executed only once per road crossing.

#### GetPosition – Action node

This node is responsible for obtaining the current GPS position of the robot and converting it to the UTM coordinate system. It is implemented as a ROS topic subscriber. The topic subscribed is `/gps/fix` where the GPS data are being published.
The obtained data are then converted to UTM using the `gps_to_utm` function defined in 4.2.3. The result is then stored as two BT blackboard variables – `easting` and `northing`.
For every obtained value, it also calls a ROS service `place_suitability` to determine the suitability of the current position for crossing.

#### CrossRoad – Condition node

This node tells our algorithm if we are close enough to a road to take over the robot's controls. If we are not the path-planning or other node is left in control.
We use the return values of the ROS service call issued in the GetPosition node. This service has two return values – `validity` and `suitability`. `Suitability` uses the road cost as well as context score to judge the place for crossing. For `validity`, we only calculate the distance of the current location to road segments from OSM.
Therefore the `validity` variable is the one determining the output of this node. The distance limit we proposed as sufficient is 10m from the center of the road.

#### PlaceSuitable – Condition node

This node states whether the current robot's location, stored as a blackboard variable, is suitable for crossing.
It uses the second return value from the ROS service called in the GetPosition node. As stated, this value takes into account the road cost for our location from the road-cost algorithm (4.2.1) and the context score calculated separately before the service call.
The context score is based on the contextual information that is available to us. This information may be passed from other nodes (e.g., computer vision node for detecting road parameters) or set by the operator.
The calculation of the context score and the process of obtaining the contextual information is described in 4.2.4.

Other nodes shown it the BT structure (fig 3.3) are currently returning `FAILURE`. These nodes are there to show the potential for further work. Their main purpose is to steer the robot to a more optimal location for crossing. In this work, we assume that the correct location was chosen in the pre-mission planning.

### 4.1.4 Perpendicular BT

The task of this tree is to position the robot perpendicular to the road. It is the second sub-tree to be executed, and as well as Init BT, it is used to prepare the robot for the crossing and is only executed once.

#### GetAzimuth – action node

This node is responsible for obtaining the robot's current azimuth. We have a ROS subscriber listening to topic published by `compass` node[1].
The compass node may publish the azimuth in several different formats. In our program, we use the ENU format in radians. But if the compass node publishes the azimuth in a different format, we have subscribers that can convert it to the desired format.
The azimuth is then stored as a blackboard variable `azimuth`.

#### RoadHeading – action node

This node calculates the heading of the closest road to the robot. We take the current robot's position from the blackboard variable `easting` and `northing` and send a request to the ROS service `get_road_heading`.
The service returns the two coordinate points representing the closest road segment's starting and ending points.
We then calculate the heading of the road segment using the function defined in this section 4.2.3.
The calculated road heading is then stored as a blackboard variable `road_heading`.

#### ComputeHeading – action node

This node uses the blackboard variable `azimuth` and `road_heading` to calculate the heading the robot should achieve to be perpendicular to the road.
The calculation is defined in section 4.2.3.
The result is stored as a blackboard variable `req_azimuth`.

#### RobotPerpendicular – condition node

This node checks if the robot is perpendicular to the road. It works by comparing the current azimuth with the required heading. Both of these values are stored in the blackboard.
We use the function defined in section 4.2.2 to compare the values to calculate the difference between two angles. The result is then compared to the threshold value, which is set to 0.1745 rad or 10°.

[1] `https://github.com/ctu-vras/compass`

#### RotateRobot – action node

This node rotates the robot to the required heading.
First, we calculate the difference between the robot's current and desired azimuth. Then, based on the difference, we proportionally set the rotation direction and speed.
The calculated movement is then published to the `cmd_vel` topic.

#### StepFromRoad – action node

If, for whatever reason, the robot is not able to rotate safely, primarily due to the possibility of ending on the road, we use this node to move the robot away from the road.
Firstly we check the difference between the robot's current azimuth and the road heading. Based on the difference, we set the direction of the movement. The movement is then published to the `cmd_vel` topic.

### 4.1.5 Crossing BT

In this tree, the main decision-making of the road crossing is located. It is the third sub-tree to be polled and the only one to be polled repeatedly.
In multiple nodes, we will use information about the detected vehicles and collision parameters for each vehicle. Therefore, we must first define the data structures used to store this information.

#### Vehicle data

The data structure for storing the information about the detected vehicles is defined in A.1.
The first struct `vehicle_info` stores the information about the single detected vehicle. The position of the vehicle is expressed in relation to the robot's frame. The robot frame means the center of the robot is the origin of the coordinate system. The $x$-axis points forward from the robot, and the $y$-axis points to the left.
The second struct `vehicles_data` stores the `vehicle_info` structs of all detected vehicles.

#### Collision data

The data structure for storing the collision parameters has the definition in A.2.
The first struct `collision_data` is used to store the collision parameters for a single vehicle.
The velocities required for the robot to make contact with the front or back of the vehicle are stored in the variables `v_front` and `v_back`, respectively. Figure 4.1 shows the contact points we calculate the velocities for. The figure is more thoroughly explained in the next part.
The `collide` variable is a boolean value that tells us if the robot will collide with the vehicle. It is calculated based on the current velocities of the robot and the vehicle.
The second struct `collisions_data` stores the information about collisions

with all detected vehicles.

### Used units

We use these units for the measured and calculated parameters:

- **Position** – meters [m]
- **Time** – seconds [s]
- **Velocity** – meters per second [$\mathrm{m\,s^{-1}}$]
- **Acceleration** – meters per second squared [$\mathrm{m\,s^{-2}}$]
- **Dimensions** – meters [m]

### Calculating the collision parameters

First, we need to state the assumptions we are making in order to simplify the calculation.
The first assumption is about the coordinate system we are using. We are using the robot's frame, where the robot's center is the system's origin, and all the positions are expressed in relation to this origin. The $x$-axis points forward, and the $y$-axis points to the left. We can assume this because the calculations are done periodically, and the results are only relevant for the current time step. It also simplifies the process, as the vehicle positions are already expressed in the robot's frame.
The second assumption is about the movement of the robot. We assume the robot is moving in a straight line with constant velocity. This is reasonable as we want the robot to be as predictable as possible, so we do not want to move the robot to the side. The assumption about the constant velocity, meaning the acceleration is zero, is also reasonable. The speeds the robot can achieve are much lower than the robot's acceleration, which can, therefore, be neglected.
The third assumption is about the movement of the vehicle. We assume the vehicle's acceleration is constant. This is a reasonable simplification as the calculation is done periodically with a high enough frequency.
The fourth assumption is that we will calculate the collision only in two dimensions. This is reasonable as the $z$-axis will not impact the occurrence of a collision. Moreover, the area over which the collision can occur is relatively small. Therefore, any terrain deviations will not significantly impact the collision.

Figure 4.1 depicts a schematic view of the collision. There are two contact points, both on the robot and the vehicle. The first one (blue) is the point where the robot is going to collide with the front of the vehicle. The second one (red) is the point where the robot is going to collide with the back of the vehicle.
For the first point, we calculate the velocity `v_front`. This velocity depicts the minimal speed of the robot to cross in front of the vehicle. We calculate the velocity `v_back` for the second point. This velocity depicts the maximal

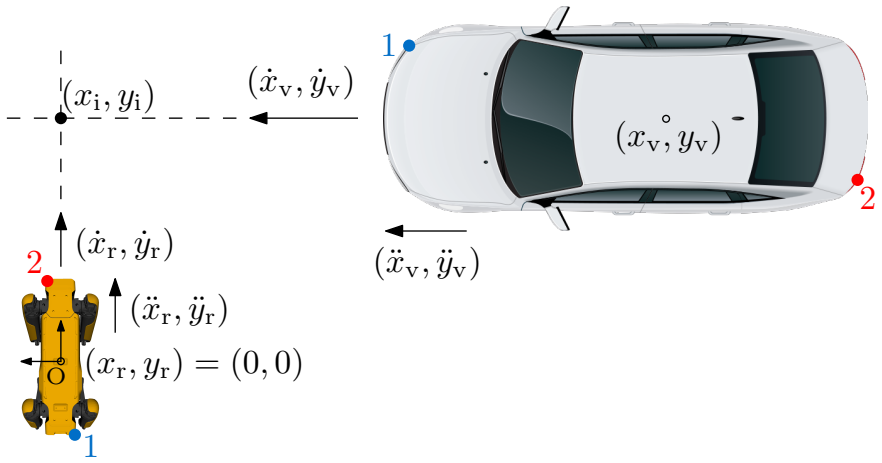



**Figure 4.1:** Visualization of collision points, coordinate system, and vehicle parameters.

speed of the robot to cross behind the vehicle.
The subscript `r` is used for the parameters of the robot, and the subscript `v` is used for the parameters of the vehicle.
The calculation is divided into three parts. In the first part, we determine the starting positions of the robot and the vehicle. In the second part, we calculate the time when the vehicle will reach the intersection point $(x_\mathrm{i}, y_\mathrm{i})$. In the last part, we calculate the velocities for the robot to collide with the vehicle.

The first part is necessary as the coordinates of both the robot and the vehicle are at the center of their respective bodies. We must move the starting points concerning the robot's and vehicle's length and width. The starting points for the robot are calculated using these equations:

$$x_{\mathrm{r,f}} = \frac{l_\mathrm{r} + w_\mathrm{v}}{2}, \tag{4.1}$$

$$x_{\mathrm{r,b}} = -\frac{l_\mathrm{r} + w_\mathrm{v}}{2}, \tag{4.2}$$

$$y_{\mathrm{r,f}} = y_{\mathrm{r,b}} = 0, \tag{4.3}$$

where $l_\mathrm{r}$ is the length of the robot and $w_\mathrm{v}$ is the width of the vehicle.
The starting points for the vehicle are calculated as follows:

$$x_{\mathrm{v,f}} = x_\mathrm{v} + \frac{l_\mathrm{v} + w_\mathrm{r}}{2} \cos(\varphi_\mathrm{v}), \tag{4.4}$$

$$x_{\mathrm{v,b}} = x_\mathrm{v} - \frac{l_\mathrm{v} + w_\mathrm{r}}{2} \cos(\varphi_\mathrm{v}), \tag{4.5}$$

$$y_{\mathrm{v,f}} = y_\mathrm{v} + \frac{l_\mathrm{v} + w_\mathrm{r}}{2} \sin(\varphi_\mathrm{v}), \tag{4.6}$$

$$y_{\mathrm{v,b}} = y_\mathrm{v} - \frac{l_\mathrm{v} + w_\mathrm{r}}{2} \sin(\varphi_\mathrm{v}), \tag{4.7}$$

where $\varphi_\mathrm{v} = \arctan\left(\frac{\dot{y}_\mathrm{v}}{\dot{x}_\mathrm{v}}\right)$ is the angle of the vehicle, $l_\mathrm{v}$ is the length of the vehicle and $w_\mathrm{r}$ is the width of the robot.
We put the width of the robot to calculate the vehicle's starting points and vice versa because we want to flatten the robot's dimensions. This is done to simplify the calculation of the intersection point of the robot's and vehicle's trajectory.
The second part of the calculation is further divided into two parts. The reason is that there are two possible scenarios for the calculation. We will use the general equation of motion [18] for both calculations.
In the first scenario, the vehicle's acceleration in the $y$-axis is zero. That means we can calculate the time using the following equations:

$$t_\mathrm{f} = -\frac{y_\mathrm{v,f}}{\dot{y}_\mathrm{v}}, \tag{4.8}$$

$$t_\mathrm{b} = -\frac{y_\mathrm{v,b}}{\dot{y}_\mathrm{v}}. \tag{4.9}$$

In the second scenario, the acceleration of the vehicle in the $y$-axis is non-zero. This scenario is more probable, as vehicles rarely drive at a constant speed. In this case, the time is calculated in the following way:

$$t_{\mathrm{f},1,2} = \frac{-\dot{y}_\mathrm{v} \pm \sqrt{\dot{y}_\mathrm{v}^2 - 2\ddot{y}_\mathrm{v} y_\mathrm{v,f}}}{\ddot{y}_\mathrm{v}}, \tag{4.10}$$

$$t_{\mathrm{b},1,2} = \frac{-\dot{y}_\mathrm{v} \pm \sqrt{\dot{y}_\mathrm{v}^2 - 2\ddot{y}_\mathrm{v} y_\mathrm{v,b}}}{\ddot{y}_\mathrm{v}} \tag{4.11}$$

There are two possible solutions for each time. The reason is that the vehicle may be decelerating and therefore change the direction of its travel. The interpretation of the results and the selection of the correct solution is discussed in the next section.
The last part of the calculation is the calculation of the velocities. First, we need to calculate the position of the vehicle in the $x$-axis at the time of the collision. We use the general equation of motion with $t_0 = 0\,\mathrm{s}$:

$$x_\mathrm{i,f} = x_\mathrm{v,f} + \dot{x}_\mathrm{v} t_\mathrm{f} + \frac{1}{2}\ddot{x}_\mathrm{v} t_\mathrm{f}^2, \tag{4.12}$$

$$x_\mathrm{i,b} = x_\mathrm{v,b} + \dot{x}_\mathrm{v} t_\mathrm{b} + \frac{1}{2}\ddot{x}_\mathrm{v} t_\mathrm{b}^2. \tag{4.13}$$

Now we can calculate the velocities of the robot.

$$\dot{x}_\mathrm{r,f} = \frac{x_\mathrm{i,f} - x_\mathrm{r,b}}{t_\mathrm{f}}, \tag{4.14}$$

$$\dot{x}_\mathrm{r,b} = \frac{x_\mathrm{i,b} - x_\mathrm{r,f}}{t_\mathrm{b}}. \tag{4.15}$$

The calculated velocities may be positive or negative. The interpretation is explained in the following section.

### Interpretation of the calculated collision parameters

We will divide this section into two parts. The first part is the interpretation of the calculated time. The second part is the interpretation of the calculated velocities.

If the calculated time is positive, the intersection point of the robot's and vehicle's trajectory is in the future. This means that the robot can collide with the vehicle without either of them changing the direction of travel.
If the calculated time is negative, it means that the intersection point of the robot's and vehicle's trajectory is in the past. This means that the robot can collide with the vehicle, but only if the vehicle or the robot would change its direction of travel.
The time can also be zero. This means that the robot and vehicle already collided. Therefore, we do not expect such time to arise as a result of the calculation.
We may have up to two solutions when calculating the times for non-zero acceleration in the $y$-axis. If we have none, the robot's and the vehicle's trajectories do not intersect.
If we have one solution, the robot's and the vehicle's trajectories intersect once. The interpretation is that the vehicle is decelerating and will stop at the intersection point and then start reversing.
If we have two solutions, the robot's and the vehicle's trajectories intersect two times. Multiple intersections could have several physical interpretations. We can interpret this as the vehicle decelerating, and therefore, changing the direction of travel after passing the intersection point. We can also interpret this as the vehicle accelerating, and therefore, the second time of the intersection is likely negative.
When choosing the calculated time, we will use the following criteria. If one time is positive and the second is negative, we will use the positive time. If both times are positive, we will use the shorter time. If both times are negative, we will use the larger time (the time that is closer to the present).

The velocities can also be positive or negative. The interpretation is similar to the one of time. Positive velocity means moving forward, while negative velocity means moving backward. The resulting velocity must also be negative if we have a negative time.
While it may seem irrelevant to calculate the time and velocity for backward movement, it is essential. This is because there is a possibility that another vehicle may be in motion, which could result in a collision with the robot. In that case, the robot will have to move backward to avoid the collision, and we need to be able to set the correct backward velocity to not collide with the first vehicle.

### GetCars – action node

The action node `GetCars` obtains information about the detected vehicles. The detection node was not yet implemented when this thesis was written.

Therefore, we will use the `GetCars` node to simulate the detection of vehicles. We will subscribe to the topic `/road_crossing/injector`. We will publish this topic from a separate node designed solely to simulate the detection node.
The information about the detected vehicles will be stored in a static variable for later use. The variable is of the format shown in listing A.1.

### CarsInTrajectory – condition node

This node is important for optimizing the flow of ticks in our behavior tree. The node is responsible for checking if there are any detected vehicles. We do not need to go through the calculation and decision-making if there are no vehicles. This speeds up the completion of this run of the BT and allows us to start the next run sooner.

### NotStarted – condition node

This node is responsible for checking if the movement across the road has been started. If it has not, we need to start the movement. If it has, we may continue with it.

### StartMovement – action node

This node is responsible for starting the movement across the road. We will set the maximal forward linear velocity to our inner static variable. This variable is responsible for storing the current velocity. The maximal velocity is chosen based on the maximal velocity of the robot used.

### MoveFwdFull – action node

This node is used when no vehicles are detected, and therefore, we want to move the robot across the road as fast as possible. We publish the maximal forward linear velocity to the topic `/cmd_vel`.

### CalculateCollision – action node

In this node, we will calculate the collision parameters. We will use the formulas described earlier. The results will be stored in the inner static variables.
This node will run the calculation for each vehicle independently. As each vehicle has its ID, we will use it to differentiate between them. This ID will also be used to delete all the results from the inner static variable when the vehicle is no longer detected.

### MoveFwd – action node

This node is used when vehicles are detected, and a forward movement is possible. We will publish the forward velocity to the topic `/cmd_vel`. The forward velocity is determined from the calculated velocities from the `CalculateCollision` node. We will set the maximal forward velocity while avoiding collisions.

### MoveBwd – action node

When the system detects the presence of other vehicles and determines that a backward movement is required, this node comes into play. To enable backward movement, we will publish the relevant velocity information to the topic `/cmd_vel`. The specific velocity will be calculated using the output from the `CalculateCollision` node. We will set the minimum backward velocity such that collisions are avoided.

### StopMovement – action node

This node is used when there is no possible movement forward without the robot colliding with a vehicle, and movement back is unnecessary or would also result in a collision. We will stop the robot by publishing zero velocity to the topic `/cmd_vel`.

### CollisionFwdMove – condition node

This node is used to determine whether there are vehicles in front of the robot in such a position and velocity that the robot would collide with them if it moved forward.

### CollisionBwdMove – condition node

Same as the previous node, this one is used to determine whether there are vehicles in such a position and velocity that the robot would collide with them if it moved backward.

### CollisionOnStop – condition node

As the two condition nodes before, this node is used to determine whether there are vehicles in such a position and velocity that the robot would collide with them if we were to stop the robot.

### CrossingFinished – condition node

In this node, we check the current position of the robot. If the distance of the current position from the middle of the road is greater than half of the width of the road, we consider the crossing finished.
Another condition for finishing the crossing is if the robot's distance from the starting point is greater than the width of the road.
Having both of these conditions is beneficial, as the robot's position may be imprecise.

## 4.2 Auxiliary functions

In this section, we will present the auxiliary functions used in the nodes of our BT algorithm.
These will include the functions used for conversions, more complex or repetitive mathematical operations, and other functions that are not directly related to the BT algorithm.
One of the big sections will be the algorithm used for determining the classi-

fication and cost of road segments in the road network.
We will split the functions into categories based on their purpose.

### 4.2.1 Road cost algorithm

We will use the algorithm developed during the summer of 2022 for the RobInGas project at the CTU CRAS. The algorithm was designed to determine the cost of crossing the road based on the road classification, curvature, and other factors.
We will briefly present the functionality of the algorithm. The full description with implementation details can be found in [19].
This is also the only part of the final solution in our thesis written in Python instead of C++ this is due to it being part of a different project. Other reasons include the usage of Python-specific libraries. While we could rewrite the code to C++, it was not deemed necessary as this part is run only once at the beginning of the mission and therefore does not need to be optimized for speed.

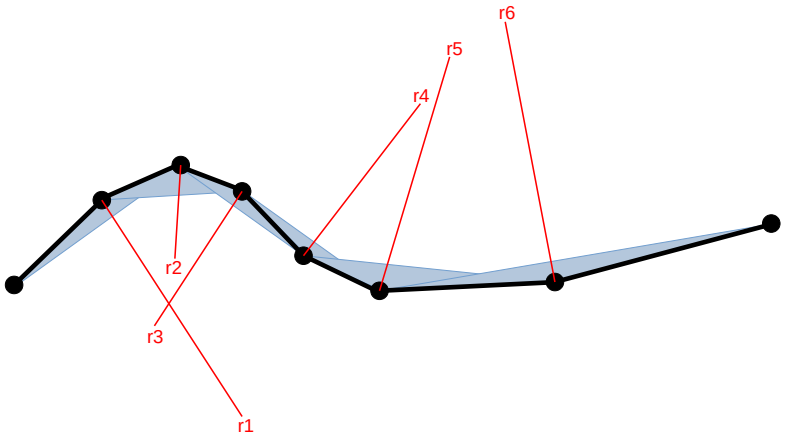
#### Overview

We use multiple parameters to determine the cost of crossing. The most important ones are the geometrical properties of the road. This includes the curvature of the road, the elevation profile, and the proximity to intersections. We also use the road classification to add to the cost function.
Other parameters would be beneficiary, such as the road width, the presence of a pedestrian crossing, and the expected traffic speed.
Unfortunately, we do not have access to all of these parameters. We use the OSM data, which does not necessarily contain all those additional parameters. Therefore, we will inject this contextual information directly into the algorithm and deal with these parameters separately. The contextual information is discussed in section 4.2.4.
The OSM data also do not contain the elevation profile of the road. Acquiring this data is not straightforward, as it is not readily available through free or open-source channels. The elevation data we use were purchased from the Land Survey Office of the Czech Republic. We use the ZABAGED [20] data. This data from the Land Survey Office are available only for the area of the Czech Republic. However, any file with elevation data with the correct formatting can be used. The file format in use conforms to the following specifications: each line of the text file contains the easting, northing, and altitude coordinates for a single point, separated by a space. Each point is described in a separate line, and lines are separated using the newline character `\n`.
If the elevation data are not provided, the algorithm will still function. It will just not take the elevation profile into account. The road cost will be determined only from the curvate, proximity to intersections, and road classification.

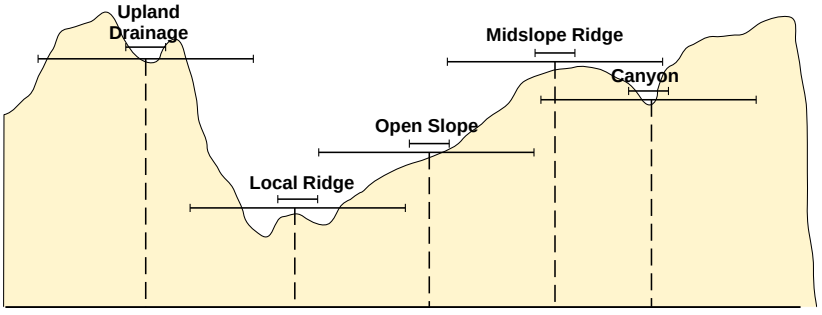
#### Algorithm

The algorithm is divided into several parts.

The first part is obtaining the road segments from downloaded OSM data. This part is also responsible for logging the road classification for each segment. The road segments from OSM data are divided into multiple smaller equidistant segments.
The second part is responsible for determining the curvature of the roads. This is done by calculating the radius of the circumcircle of the triangle formed by the two adjacent road segments. This approach is visualized in image 4.2a. We then sort the road segments into multiple classes based on their radius. In this part, we also detect intersections and penalize the road segments close to them.
In the third part, we determine the elevation profile of the road. We then classify the road segments using the TPI (Terrain Profile Index) method. Some TPI classes are presented in image 4.2b.
In the last part, we combine the results from the previous parts and calculate the final cost of crossing for each segment. These costs are then saved to a file to be used later.




**(a) :** Visualization of the circumcircle radii.




**(b) :** Representation of certain TPI classes.

**Figure 4.2:** Visualization of key elements in the road cost algorithm.

### Usage

As was stated earlier, this algorithm is executed only once at the beginning of the mission. Later we only keep the final costs, and based on them, we determine if the location where the robot is trying to cross is suitable and

safe.
We rely on ROS to enable the communication between our main algorithm and the algorithm for determining the suitability of the location for crossing. We implemented a ROS service for this purpose.
Another part of the algorithm that could be implemented in future work is to try and provide the robot with a more suitable crossing place. This could be used if the pre-mission planning was not performed or the robot's path changed, and the current location is unsuitable. If such a place is found and provided, a cost map that will be used to change the current cost map of the path planner should be published. This is done so that the robot's controls are not overridden until we begin the crossing itself.

### 4.2.2 Mathematical functions

#### Difference between two angles

This function is used to determine the difference between two given angles. We assume the angles given are in radians, and both are in the interval $\langle 0; 2\pi \rangle$. This difference is calculated to be the smallest possible and to fit within the interval $\langle -\pi; \pi \rangle$. The formula we use is a modified version of the one provided here [21] and has the following form

$$\Delta\varphi = ((\varphi_2 - \varphi_1 + \pi) \mod (2\pi)) - \pi. \tag{4.16}$$

Before returning the result, we check whether the result is within the specified interval.

#### Converting degrees to radians

While this function is elementary in its nature, it is frequently used in our code. Therefore it is beneficiary to create this function.
The equation for converting degrees to radians which this function uses is as follows

$$\varphi_{\mathrm{RAD}} = \varphi_{\mathrm{DEG}} \frac{\pi}{180}. \tag{4.17}$$

### 4.2.3 Geographical functions

Here we will present the functions used for geographical calculations. These include conversions between coordinate systems, calculating azimuths, and others.

#### Converting GPS to UTM

While most geographical data are stored in the WGS84 coordinate system, generally known as GPS, the UTM coordinate system is more suitable for calculations. Therefore, we will convert the GPS coordinates to UTM.
When working with geographical conversions in C++, we use the library GeographicLib. The function from this library that provides the conversion is `GeographicLib::UTMUPS::Forward`. This function takes the point's latitude and longitude and returns the point's easting and northing in the UTM

coordinate system. It also returns the zone number and whether the point is in the northern or southern hemisphere.
While the input variables are passed by value, the return variables are passed to the function call by reference.
We use the `utm` library when converting geographical data in Python. The function facilitating the conversion from WGS84 to UTM is `utm.from_latlon`.

### Converting NED to ENU

There are two possible orientations of an azimuth. The NED (North-East-Down) and the ENU (East-North-Up).
NED means that azimuth 0 points north, and its value increases clockwise. This orientation is mainly used in cartography and everyday life.
ENU means that azimuth 0 points east, and its value increases clockwise. This orientation is mainly used in navigation and robotics, as it is consistent with REP-103 [22].
In the entire project, we use the ENU orientation. However, as we rely on other ROS nodes to provide us with the azimuth, we need to be able to convert the azimuth from NED to ENU.
The conversion should be much simpler since we do not deal with coordinates but with already computed azimuths.
The image 4.3 shows the two possible orientations of the azimuth. This image also provides us with the insight we need to determine the conversion formula.
We need to divide the formula into two parts.
The first option is when the azimuth (in NED) is between 0 rad and $\frac{\pi}{2}$ rad.

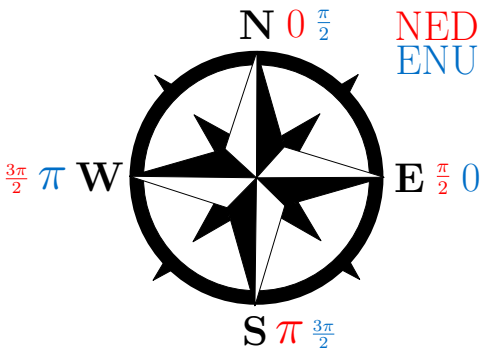



**Figure 4.3:** Two possible orientations of the azimuth.

In this case, the azimuth in ENU is computed in the following way

$$a_{\text{ENU}} = \frac{\pi}{2} - a_{\text{NED}}, \tag{4.18}$$

where $a_{\text{ENU}}$ is the azimuth in ENU and $a_{\text{NED}}$ is the azimuth in NED.
The second option is for all other azimuths, e.g., when the azimuth is between $\frac{\pi}{2}$ rad and $2\pi$ rad. In this case, the azimuth in ENU is computed in the following way

$$a_{\text{ENU}} = \frac{5\pi}{2} - a_{\text{NED}}. \tag{4.19}$$

#### Compute azimuth from coordinates

This function is used to compute the azimuth for an observer standing at the first point and looking at the second point.
We will use a slightly modified version of the formula provided here [23]. This calculation was created for the WGS84 coordinate system, however, we use the UTM coordinate system. Having said that, we can use the same formula, as the WGS84 to UTM projection is conformal [24]. In testing the difference between calculated azimuths using the WGS84 and UTM coordinates was around $\Delta\varphi = 0.097\,\text{rad}$ or $\Delta\varphi = 5.541°$.
The computational equations are as follows

$$\Delta y = y_2 - y_1, \tag{4.20}$$

$$\alpha = \sin(\Delta y)\,\cos(x_2), \tag{4.21}$$

$$\beta = \cos(x_1)\,\sin(x_2) - \sin(x_1)\,\cos(x_2)\,\cos(\Delta y), \tag{4.22}$$

$$\varphi = \arctan\left(\frac{\alpha}{\beta}\right), \tag{4.23}$$

$$\varphi = (\varphi + 2\pi) \mod (2\pi). \tag{4.24}$$

Where $x_1$ and $y_1$ are the latitude and longitude of the first point, and $x_2$ and $y_2$ are the latitude and longitude of the second point.

#### Compute heading for robot

The use of this function is to determine the heading of the robot. It is used to get the robot perpendicular to the road.
The function takes the robot's azimuth and the road's heading. The algorithm creates two new variables, one $+\frac{\pi}{2}$ and one $-\frac{\pi}{2}$ from the road's heading. This is necessary as we do not know in what order road points are stored, and we do not need to differentiate the side we approach the road from.
Then it computes the difference between the robot's heading and the two new azimuths. The smaller difference is then returned.

### 4.2.4 Contextual information and score

#### Contextual information

Contextual information provides us with valuable information about the environment. This information is a vital part of choosing the best location for crossing.
The contextual information we use is the following

- **Maximal speed** – The maximal speed of vehicles on the road.
- **Number of lanes** – The number of lanes on the road.
- **Road width** – The width of the road.
- **Road type** – The type of the road.
- **Pedestrian crossing** – Whether there is a pedestrian crossing on the road and its location.

### Obtaining contextual information

Contextual information may be obtained from several sources. One source could be the OSM database. However, this database is not always up to date, and there is no guarantee that the necessary information will be available. Other sources could be the direct observations of the environment or the information provided by other ROS nodes.
In our case, the operator will submit the information. We have prepared a Python class with the appropriate variables and functions. To use the contextual information, the operator should create an instance of this class and fill in the appropriate variables.
The creation of the class is recommended to be done in advance. It can be automated or done manually. The class also contains functions necessary for saving and loading contextual information to a file.
During the execution of our algorithm, the tree node will call a ROS service to obtain the contextual information. This service will return the contextual information for the closest road to the requested location.
Thanks to this approach, we are able to log the contextual information for multiple roads and use them to evaluate the crossing locations.

### Calculating the context score

The score is calculated as a sum of points for each contextual information.

$$\xi_{\text{context}} = \sum_{i=1}^{n} \xi_{\text{context,i}}, \tag{4.25}$$

where $n = 5$ as we currently have five different types of contextual information. The individual points are set as follows

- **Maximal speed** – $v_{\max}$
  - $v_{\max} \leq 30 \quad \rightarrow \quad \xi_{\text{context},1} = 3$
  - $v_{\max} \leq 50 \quad \rightarrow \quad \xi_{\text{context},1} = 2$
  - $v_{\max} \leq 80 \quad \rightarrow \quad \xi_{\text{context},1} = 1$
- **Number of lanes** – $n_{\text{lanes}}$
  - $n_{\text{lanes}} = 1 \quad \rightarrow \quad \xi_{\text{context},2} = 5$
  - $n_{\text{lanes}} = 2 \quad \rightarrow \quad \xi_{\text{context},2} = 4$
  - $n_{\text{lanes}} = 3 \quad \rightarrow \quad \xi_{\text{context},2} = 2$
  - $n_{\text{lanes}} = 4 \quad \rightarrow \quad \xi_{\text{context},2} = 1$
- **Road width** – $w_{\text{road}}$
  - $w_{\text{road}} \leq 3.5 \quad \rightarrow \quad \xi_{\text{context},3} = 4$
  - $w_{\text{road}} \leq 4.5 \quad \rightarrow \quad \xi_{\text{context},3} = 3$
  - $w_{\text{road}} \leq 5.5 \quad \rightarrow \quad \xi_{\text{context},3} = 2$
  - $w_{\text{road}} \leq 6.5 \quad \rightarrow \quad \xi_{\text{context},3} = 1$

- **Road type**
  - motorway → $\xi_{\text{context},4} = -10$
  - trunk → $\xi_{\text{context},4} = -4$
  - primary → $\xi_{\text{context},4} = 1$
  - secondary → $\xi_{\text{context},4} = 2$
  - tertiary → $\xi_{\text{context},4} = 3$
- **Pedestrian crossing**
  - if present → $\xi_{\text{context},5} = 10$

## 4.3 ROS-specific functions

Here we will present the ROS-specific parts of our algorithm. These include the ROS nodes, services, and messages.

### 4.3.1 ROS services

We use multiple ROS services in our algorithm. These services are mostly used to obtain information about the road we are crossing. All services are implemented within the `road_crossing` package.
We will present the individual services and their purpose in the algorithm. However, we will not show the definition of the service messages or go into details about the implementation of the services.

#### GetFinish

This service is used to determine whether the robot's current position is optimal for finishing the crossing.
This service is used by the `CrossingFinished` node of the `Crossing` sub-tree. The determining factor is explained in the `CrossingFinished` node section.

#### GetRoadInfo

This service is used to obtain information about the road we are crossing. This information includes the position of the road, contextual information, and the starting position of the robot. The position of the road is set by the two points defining the road segment we used in the road cost algorithm. The contextual information contents were described earlier. The robot's starting position was the geographical position of the robot when the crossing algorithm started.
The service call takes the position of the robot in the UTM format. It then finds the closest road segment from the saved segments with information. The final distance to the road segment does not matter; we pick the closest one. It then returns the information about the road segment.

#### GetRoadSegment

This service is used to obtain the road segment the robot is crossing. The road segment's definition was presented in the previous service.
The service call takes the position of the robot in the UTM format. It then finds the closest road segment from the saved segments from the road cost algorithm. The location of the two defining points of the road segment is then returned.

#### GetSuitability

This service is used to determine if the position of the robot is valid and suitable for crossing the road.
The service call takes the geographical position of the robot in the UTM format and the contextual score. It finds the closest road segment from the road cost algorithm results. If the distance to any road segment is greater than 10 meters, the position is deemed invalid for crossing, and such is returned. If the position is valid, it then calculates the final cost score based on the cost of the road segment and the contextual score. The position is deemed unsuitable for crossing if the final score is below a set threshold. The threshold is set to 20.

#### StartAlgorithm

This service is used to start and end the crossing algorithm. This service is intended to be used from the outside of the algorithm.
The call takes two boolean values, start and stop. Only one is permitted to be set as true. Otherwise, a warning is raised.

### 4.3.2 ROS nodes and messages

Our algorithm uses several ROS nodes, primarily service servers. The main crossing algorithm is executed by a dedicated node, while other non-server nodes simulate the operation of various ROS projects. For example, some nodes handle vehicle detection simulation, while others initiate the start or stop of the algorithm. All nodes are encapsulated within the `road_crossing` package.
The communication between nodes is provided via ROS topics with custom ROS messages.

#### Injector messages

We use these messages to emulate the behavior of the vehicle detection node, as they enable us to input vehicle information into the algorithm. Specifically, they simulate the data that would typically be obtained from a 3D bounding box generated from LiDAR data. The data they simulate is later parsed and stored as a `vehicle_info` object from A.1.

#### Start messages

These messages communicate the change in the run state of the main algorithm. This is necessary as the start/stop service is in a different node,

making the state variable inaccessible from the main algorithm.

### Service nodes

To implement the services detailed in the previous section, we created separate nodes for each. Each node was designed as a service server. We opted to implement the nodes using Python, as they depend on data generated by the road cost algorithm and require access to the data structures used to store the costs.

### Crossing algorithm node

The primary algorithm developed in this thesis is executed using a single node. This node is responsible for subscribing to the relevant topics required for the algorithm's execution, including `/gps/fix`, `/compass/...`, `/road_crossing/start`, and `/road_crossing/injector`. The specific compass topic is determined dynamically during the algorithm's runtime based on the available compass topics. Additionally, the node initializes all of the required publishers and server clients.
For each iteration of the main loop within this node, we perform a single round of ROS callbacks and tick the behavior tree.

### Launch files

Within our package, we have three launch files. The first, `services.launch`, is responsible for initializing and running the service servers. Alongside this, it launches two other nodes. The `get_mag_shift` and `magnetometer_compass` from `gps_to_path` and `compass` packages respectively.
The second launch file, `crossing.launch`, is used to launch the primary algorithm node.
Finally, the `demo.launch` file combines the previous two launch files, making it possible to execute both the service servers and the primary algorithm node in one launch command.

# Chapter 5

# Simulation experiments

The main goal of the simulation experiments was to verify the functionality of the algorithm design. The second goal was to improve the algorithm and its nodes. The third goal was to create several scenarios for the algorithm to test its behavior in different situations. These scenarios were created with later real-world experiments in mind.

## 5.1 Algorithm functionality experiments

The first experiments were created to test the execution capabilities of the algorithm. They aimed to test if the nodes would not crash or stall the system. These experiments helped to find and fix multiple bugs in the nodes.
Moreover, we were able to detect some weak spots in the BT structure and improve it. The `Groot` application's log viewer was an invaluable tool for detecting these weak spots. The screen of the log viewer is shown in 5.1.

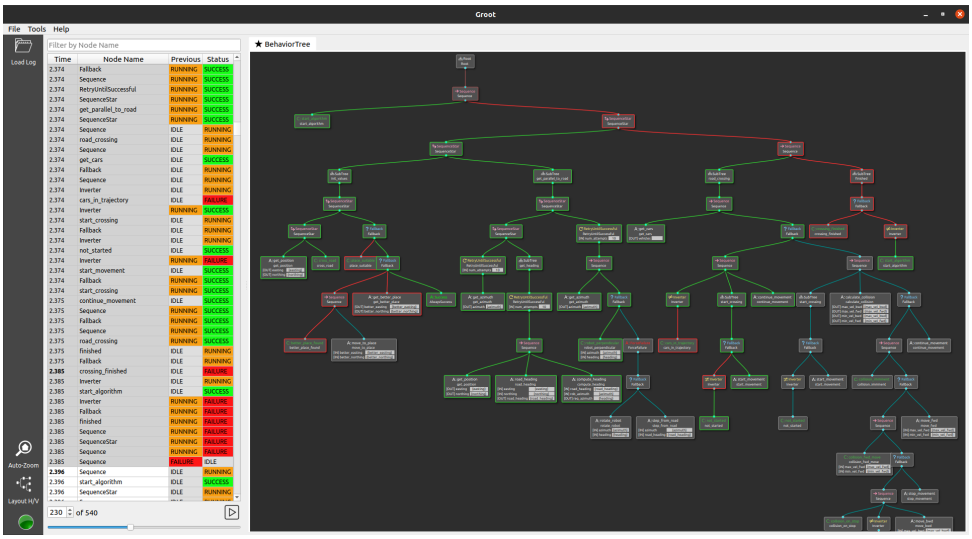



**Figure 5.1:** Log viewer in `Groot` application.

## 5.2 Algorithm behavior experiments

Once the basic functionality was verified, we created several scenarios to test the behavior and universality of the algorithm. This was done to verify that the algorithm would work in different situations.

# Appendix A

## Data structures

**Listing A.1:** Vehicle data structure

```
struct vehicle_info {
    int id;
    double x_pos;
    double y_pos;
    double x_dot;
    double y_dot;
    double x_ddot;
    double y_ddot;
    double length;
    double width;
};
struct vehicles_data {
    int num_vehicles;
    std::vector<vehicle_info> data;
};
```

**Listing A.2:** Collision data structure

```
struct collision_info {
    int car_id;
    double v_front;
    double v_back;
    bool collide;
    bool collide_stop;
};
struct collisions_data {
    int num_collisions;
    std::vector<collision_info> data;
};
```

# Appendix B

## Bibliography


[1] M. Colledanchise and P. Ögren. *Behavior Trees in Robotics and AI: An introduction*. CRC Press, 2018. ISBN: 9781138593732.

[2] Mohamad Bdiwi et al. "Towards safety4.0: A novel approach for flexible human-robot-interaction based on safety-related dynamic finite-state machine with multilayer operation modes". In: *Frontiers in Robotics and AI* 9 (Sept. 2022), p. 1002226. DOI: 10.3389/frobt.2022.1002226.

[3] Richard Balogh and David Obdržálek. "Using Finite State Machines in Introductory Robotics: Methods and Applications for Teaching and Learning". In: Jan. 2019, pp. 85–91. ISBN: 978-3-319-97084-4. DOI: 10.1007/978-3-319-97085-1_9.

[4] Magnus Olsson. "Behavior Trees for decision-making in Autonomous Driving". In: 2016.

[5] Matteo Iovino et al. "A survey of Behavior Trees in robotics and AI". In: *Robotics and Autonomous Systems* 154 (2022), p. 104096. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2022.104096. URL: https://www.sciencedirect.com/science/article/pii/S0921889022000513.

[6] OpenStreetMap Wiki. *Main Page — OpenStreetMap Wiki,* [Online]. 2022. URL: https://wiki.openstreetmap.org/w/index.php?title=Main_Page.

[7] voretaq7 (https://aviation.stackexchange.com/users/64/voretaq7). *What is the difference between azimuth and heading?* Aviation Stack Exchange. (version: 2017-04-13). URL: https://aviation.stackexchange.com/a/24901.

[8] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[9] Razan Ghzouli et al. *Behavior Trees and State Machines in Robotics Applications*. 2022. DOI: 10.48550/ARXIV.2208.04211. URL: https://arxiv.org/abs/2208.04211.

[10] Davide Faconti and Eurecat. *BehaviorTree.CPP documentation*. Version 3.8. 2022. URL: https://behaviortree.github.io/BehaviorTree.CPP/.

[11] Auryn Robotics. *BehaviorTree.CPP documentation*. Version 4.0.1. 2023. URL: https://www.behaviortree.dev/.

[12] OpenStreetMap Wiki. *Overpass API — OpenStreetMap Wiki,* [Online]. 2023. URL: https://wiki.openstreetmap.org/w/index.php?title=Overpass_API.

[13] Sean Gillies. *The Shapely User Manual*. Version 2.0.1. 2023. URL: https://shapely.readthedocs.io/en/stable/manual.html.

[14] NumPy Developers. *NumPy documentation*. Version 1.24. 2022. URL: https://numpy.org/doc/stable/.

[15] Charles F. F. Karney. *GeographicLib documentation*. Version 2.2. 2023. URL: https://geographiclib.sourceforge.io/C++/doc/index.html.

[16] Thinal Raj et al. "A Survey on LiDAR Scanning Mechanisms". In: *Electronics* 9.5 (2020). ISSN: 2079-9292. URL: https://www.mdpi.com/2079-9292/9/5/741.

[17] Bikramjit Banerjee. "Autonomous Acquisition of Behavior Trees for Robot Control". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 3460–3467. DOI: 10.1109/IROS.2018.8594083.

[18] Michal Bednařík. *Fyzika 1 pro Kybernetiku a robotiku*. Equation 5.43. skripta ČVUT FEL, 2021, p. 36.

[19] Jan Vlk. *Road crossing cost algorithm*. [Online]. 2022. URL: https://github.com/ctu-vras/gps-navigation/blob/dev_road_crossing/osm_path/scripts/Road%20crossing%20docs/road_crossing_algorithm.pdf.

[20] *ZABAGED – planimetric components – introduction*. https://geoportal.cuzk.cz/(S(dcpfei0nmxcwgoe4frurwfgm))/Default.aspx?lng=EN&mode=TextMeta&text=dSady_zabaged&side=zabaged&menu=24. Accessed: 2023-03-20.

[21] Kyle (https://stackoverflow.com/users/2779530/kyle). *Finding the shortest distance between two angles*. Stack Overflow. (version: 2021-04-08). URL: https://stackoverflow.com/a/28037434.

[22] Tully Foote and Mike Purvis. *Standard Units of Measure and Coordinate Conventions*. https://www.ros.org/reps/rep-0103.html. 2010.

[23] Nayanesh Gupte (https://stackoverflow.com/users/1000864/nayanesh-gupte). *Calculate angle between two Latitude/Longitude points*. Stack Overflow. (version: 2018-03-20). URL: https://stackoverflow.com/a/18738281.

[24] John P. Snyder. *Map projections: A working manual*. U.S. Government Printing Office, 1987, p. 48. DOI: 10.3133/pp1395.