UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2023-085
The Faculty of Science, Technology and Medicine

# DISSERTATION

Defence held on 29/09/2023 in Esch-sur-Alzette

to obtain the degree of

# DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

# EN INFORMATIQUE

by

Samira CHAYCHI
Born on 22 October 1987 in Maragheh, Iran

# PROACTIVE COMPUTING PARADIGM APPLIED TO THE PROGRAMMING OF ROBOTIC SYSTEMS

## Dissertation defence committee

Prof. Dr. Denis ZAMPUNIERIS, dissertation supervisor
*Professor, Université du Luxembourg*

Prof. Dr. Martin THEOBALD
*Professor, Université du Luxembourg*

Prof. Dr. Steffen ROTHKUGEL, Chairman
*Professor, Université du Luxembourg*

Dr. Eric Wagner
*Research Associate, Universität des Saarlandes*

Prof. Dr. Jean-Noël COLIN, Vice Chairman
*Professor, Université de Namur*

# Declaration of Authorship

I, Samira CHAYCHI, declare that this thesis titled, “Proactive Computing Paradigm Applied to the Programming of Robotic Systems” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

*"The best way to predict the future is to invent it."*

Alan Kay

UNIVERSITY OF LUXEMBOURG

# *Abstract*

Doctor of Philosophy

**Proactive Computing Paradigm Applied to the Programming of Robotic Systems**

This doctoral thesis is concerned with the development of advanced software for robotic systems, an area still in its experimental infancy, lacking essential methodologies from generic software engineering. A significant challenge within this domain is the absence of a well-established separation of concerns from the design phase. This deficiency is exemplified by Navigation 2, a real-world reference application for (semi-) autonomous robot journeys developed for and on top of the Robot Operating System (ROS): the project's leading researchers encountered difficulties in maintaining and evolving their complex software, even for supposed-to-be straightforward new functions, leading to a halt in further development. In response, this thesis first presents an alternative design and implementation approach that not only rectifies the issues but also elevates the programming level of consistent robot behaviors. By leveraging the proactive computing paradigm, our dedicated software engineering model provides programmers with enhanced code extension, reusability and maintenance capabilities. Furthermore, a key advantage of the model lies in its dynamic adaptability via on-the-fly strategy change in decision-making. Second, in order to provide a comprehensive evaluation of the two systems, an exhaustive comparative study between Navigation 2 and the same application implemented along the lines of our model, is conducted. This study covers thorough assessments at both compile-time and runtime. Software metrics such as coupling, lack of cohesion, complexity, and various size measures are employed to quantify and visualize code quality and efficiency attributes. The CodeMR software tool aids in visualizing these metrics, while runtime analysis involves monitoring CPU and memory usage through the Datadog monitoring software. Preliminary findings indicate that our implementation either matches or surpasses Navigation 2's performance while simultaneously enhancing code structure and simplifying modifications and extensions of the code base.

# *Acknowledgements*

I am grateful for the opportunity to express my sincere appreciation to Professor Dr. Denis Zampunieris for his invaluable support and guidance over the past few years. As a member of his research and teaching team, I have had the privilege of gaining profound insights from his extensive knowledge and expertise in our field. His constructive feedback and encouragement have been invaluable in helping me to grow and develop as a researcher. I am truly thankful for his time and dedication to my academic and professional success.

I would like to take this opportunity to express my gratitude to Sandro Reis and the members of my CET, professors Martin Theobald and Jean-Noël Colin, for their invaluable advice and guidance during the past few years. Their support and encouragement have been instrumental in helping me to navigate the challenges of research and learning. Their expertise and diverse perspectives have contributed greatly to the success of our collaborative efforts. In addition, I express my gratitude to the respected professor Steffen Rothkugel and Dr. Eric Wagner, for kindly accepting to be part of my thesis committee.

Lastly, I would like to express my heartfelt appreciation to my parents, who have been my pillars of strength throughout this journey. Their unwavering love, encouragement, and support have kept me going, and I could not have accomplished this without them. I would also like to thank my close friends for their constant moral support, which has continually inspired and focused my efforts.

# Contents

# List of Figures

# List of Abbreviations

| | |
|---|---|
| **ROS** | **R**obot **O**perating **S**ystem |
| **PE** | **P**roactive **E**ngine |
| **SoC** | **S**eparation **o**f **C**oncerns |
| **FIFO** | **F**irst **I**n **F**irst **O**ut |
| **DDS** | **D**ata **D**istribution **S**ervice |
| **RTPS** | **R**eal-time **P**ublish **S**ubscribe Protocol |
| **BT** | **B**ehavior **T**ree |
| **ODE** | **O**pen **D**ynamics **E**ngine |
| **DM** | **D**ecision **M**aking |
| **CBO** | **C**oupling **B**etween **O**bject |
| **LCOM** | **L**ack of **C**ohesion **O**f **M**ethods |
| **WMC** | **W**eighted **M**ethod **C**ount |
| **RFC** | **R**esponse **F**or a **C**lass |
| **DIT** | **D**epth of **I**nheritance **T**ree |
| **LOC** | **L**ine **O**f **C**ode |
| **NOM** | **N**umber **O**f **M**ethods |
| **C3** | **C**oupling **C**ohesion **C**omplexity |
| **CPU** | **C**entral **P**rocessing **U**nit |

# Chapter 1

# Introduction

In the world of software applications, a fundamental challenge has long plagued developers in the realm of robotic systems: the absence of a robust software development methodology. Many existing applications lack the flexibility required for seamless adaptation, reuse, and maintenance fundamental pillars of software engineering.
To address these pressing issues, the concept of Separation of Concerns (SoC)[1] emerges as a promising solution. SoC is a methodology that advocates dividing computer programs into distinct sections, with each section addressing a specific concern or set of information related to the program's code. By applying SoC, software developers gain greater freedom to simplify and maintain their code. This approach results in what is referred to as modular programming, offering enhanced opportunities for individual modules to evolve, be reused, and developed independently. This, in turn, facilitates the extension and maintenance of the code base.
Our investigation aims to discover innovative solutions for creating a more modular and maintainable software landscape. Additionally, we explore the potential of the proactive computing paradigm[Tennenhouse, 2000], which is realized through a rule-based proactive engine, as a coding approach to address our first research question. This exploration seeks to determine whether this paradigm can serve as an effective foundation for building more efficient and organized robotic software systems. Furthermore, our research delves into the intricacies of managing conflicts within proactive decision-making scenarios. These scenarios often involve conflicting recommendations stemming from multiple independent objective-based scenarios. We are actively seeking strategies and methodologies to effectively navigate these complexities. Lastly, we contemplate the design and implementation of smart and self-managing behavior within robotic systems. This inquiry revolves around enabling dynamic changes in strategy, paving the way for more adaptable and

[1] https://en.wikipedia.org/wiki/Separation_of_concerns

responsive robotic software.
Based on these challenges and research questions, our thesis aims to demonstrate the advantages of employing separation of concerns principles to overcome existing hurdles within robotic software systems. The core principle of SoC is to facilitate the creation of a well-organized model comprising independent components, each addressing a distinct concern. To embody this SoC principle, we propose a model for developing robotic software systems using a Proactive Engine. The Proactive Engine represents the implementation of a rule-based proactive system, harnessing the strengths of object-oriented principles and rule-based systems. It consists of a rule engine, a database, and rules. These rules can be categorized into scenarios, which are sets of rules executed in response to specific events. This engine serves as a middleware system that can seamlessly integrate with other systems [Chaychi, Zampunieris, and Reis, 2023].
Within the Proactive Engine, scenarios play a pivotal role—a scenario comprises a combination of proactive rules, with each rule responsible for a specific action. Scenarios exhibit a wide range of features, structures, and complexities, making them adaptable to various domains and situations. In our project, we utilize the Robotic Operating System (ROS) [M and al, 2009] to implement robot simulations. ROS, an open-source operating system, facilitates message passing between processes, package management, device control, and more. It emphasizes code reusability, extensibility, and maintenance in the realm of robotics. This doctoral thesis endeavors to provide insights into enhancing robotic software engineering by embracing SoC principles and the innovative use of a Proactive Engine within the ROS ecosystem. Our goal is to contribute to the evolution of intelligent, adaptable, and resilient robotic systems, paving the way for future advancements in this field.

## 1.1 Research Questions

The inspiration for our proposed model was sparked by a proof-of-concept paper featuring a virtual robot deployed in the Webots™ simulator [Frantz and Zampunieris, 2020]. This concept led us to formulate several critical research questions, including:

1. How to improve separation of concerns in robotic software engineering? How can software metrics be used to measure the enhancement?

2. Is the proactive computing paradigm, implemented through a rule-based proactive engine, an adequate coding approach for addressing our first research question?

3. How can conflict handling be effectively managed in proactive decision-making scenarios where conflicting recommendations arise from multiple independent objective-based scenarios?

4. How can smart and self-managing behavior be designed and implemented in the system to address our third research question, enabling dynamic changes in strategy?

## 1.2 Thesis Structure

After the introduction section, the subsequent chapter will focus on the problem statement. This chapter emphasizes the critical absence of separation of concerns in robotic software systems. It engages in a comprehensive discussion about the far-reaching consequences of this absence on the domain of robotics technology. Furthermore, it provides a comprehensive overview of the contemporary landscape of robotics and the principle of Separation of Concerns. The chapter also squarely addresses the multifaceted challenges associated with the complexity of robotic systems, delving into the intricate ramifications that stem from the lack of proper separation of concerns. Notably, the chapter shines a spotlight on the challenges pertaining to modification, reusability, and maintenance within this context.
Following this, we will delve into a dedicated tools chapter. This section will elaborate on the arsenal of tools that were instrumental in materializing our research model. The tools utilized encompass the ROS and Navigation 2 [Macenski et al., 2020] as a case study, a Proactive Engine, simulation and visualization tools, and a robust database system. By detailing the utilization of these tools, we demonstrate their role in shaping the implementation and methodology of our research.
The proposed model chapter will provide a comprehensive exploration of our innovative model. It will elucidate the fundamental issue of inadequate separation of concerns, accentuating the significance of attributes like modifiability, reusability, and maintainability in robotic software systems. Central to this chapter will be a heightened focus on the enhancement of navigation systems, bolstered by the introduction of the Proactive Engine. This dynamic element, the Proactive Engine, emerges as a pivotal strategy for achieving

improved separation of concerns. The chapter will also intricately describe different scenarios within the Proactive Engine and provide an overview of how our proposed model seamlessly integrates with ROS for better modularity and performance.

Transitioning into the subsequent chapter dedicated to implementation, intricate details will be unveiled. The architecture forging a connection between ROS and the Proactive Engine through a MySQL database will be intricately described. Real-time data and command exchange mechanisms will be laid bare, along with the code snippets that exemplify the operation of individual scenarios within the Proactive Engine. The implementation design within ROS, comprising subscriber and publisher nodes, will be meticulously explained. Notably, the design choices that enable efficient data management and communication between robotic systems and the database will be highlighted.

A chapter of comparison follows, where we undertake a thorough analysis to juxtapose our proposed solution against Navigation 2. This analysis will be multifaceted, encompassing aspects such as compile time, runtime, system modification, extension capabilities, and responsiveness to dynamic changes. For these analyses, we will employ sophisticated tools like CodeMR and Datadog.

Lastly, the conclusion chapter will encapsulate the entire journey. It will revisit the research questions and objectives, encapsulate the insights to be garnered, and highlight the significant contributions to be made to the domain of robotics software engineering. Reflecting on the implications of the proposed solutions, the chapter will acknowledge limitations and articulate areas for potential refinement. Furthermore, it will map out avenues for future research endeavors, particularly focusing on enhancing the concept of separation of concerns within the realm of robotics.

# Chapter 2

# Problem Statement

## 2.1 Introduction

In this chapter, we delve into the problem statement at the heart of our research, addressing the profound impact of the absence of separation of concerns on robotic software systems. We explore the challenges it poses, such as complexity, codebase maintainability, and hindered advancements in robotics technology. By unraveling these intricacies, we not only lay the foundation for innovative solutions and design principles but also set the stage for our exploration of novel strategies. These strategies aim to revolutionize the way robotic software is conceptualized, developed, and maintained, ultimately paving the way for a more efficient, modular, and scalable future in robotics software engineering.

This exploration is complemented by an overview of the current state of the art concerning the Navigation 2 and the concept of Separation of Concerns. Alongside this, we delve into the challenges arising from the absence of proper separation of concerns within systems. The central challenges we address in this thesis are centered around modification, reusability, and maintenance. In subsequent sections, we delve deeper into these challenges and present our proposed solutions, aiming to effectively tackle them. The work we present here aims to make a significant contribution to the advancement of robotics software engineering, fostering a more scalable and adaptable approach to the development of robotic systems.

## 2.2 Separation of Concerns

Separation of concerns (SoC) is a well-established software engineering principle that advocates for the decomposition of a system into distinct parts, each addressing a separate and well-defined concern. The ultimate objective of SoC is to create a modular system where each module or component

has a clear, singular responsibility, enhancing the system's comprehensibility, maintainability, and extensibility. By segregating different concerns into separate components, modifying or extending the system becomes more manageable without affecting other parts. This approach reduces the risk of unintended consequences and simplifies the debugging and troubleshooting process[1].

The concept of SoC has gained widespread recognition and extensive discussion in various software development resources, such as books, papers, and articles within the fields of computer science and software engineering [ Heckel and Engels, 2002, Andrade et al., 2002]. SoC refers to the practice of tackling a program's complexity by separating its fundamental computational algorithms, making the software more manageable. This idea has given rise to a new research area and software development paradigm known as "aspect-oriented software development" [Elrad, Filman, and Bader, 2001]. Aspect-Oriented Programming (AOP) is a technology that supports the separation of concerns in software engineering. Consequently, it becomes evident that applying a technology designed to handle distinct concerns within a single application can significantly contribute to usability engineering. Remote usability testing has proven to deliver promising results, and AOP provides the ideal approach to streamline the testing process for various software products without mixing concerns. By separating the generation of test data from program execution [Elrad, Filman, and Bader, 2001], AOP empowers developers and testers to focus on specific aspects of the software independently, leading to more effective and efficient testing procedures.

One of the earliest references to the concept of SoC can be traced back to Edsger W. Dijkstra's influential 1968 paper [Dijkstra, 2001]. In this groundbreaking work, Dijkstra highlights the significance of structured programming and the separation of concerns as fundamental elements in elevating software quality. Dijkstra's emphasis on structured programming underscores the significance of organizing code into well defined modules, facilitating code readability, maintainability, and reducing potential errors. Additionally, his recognition of the separation of concerns as a crucial principle addresses the need to isolate different aspects of a software system, allowing

[1] https://en.wikipedia.org/wiki/Separation_of_concerns

developers to focus on individual functionalities independently. By incorporating these principles into software development practices, Dijkstra's insights have played a pivotal role in shaping modern programming methodologies. The lasting impact of his work continues to influence the way software engineers approach problem-solving and the design of complex systems, ultimately leading to the creation of more efficient, reliable, and scalable software solutions.

Based on [Heckel and Engels, 2002], the functional and non-functional requirements are initially separated into distinct sub-models during the early stages of development. However, when these sub-models need to be integrated into a unified system, consistency issues often arise between them. Moreover, any changes made to the sub-models can exacerbate these problems. To address these challenges, the paper proposes a relational approach that pairs functional and non-functional models, utilizing meta-models as an additional layer of abstraction. This technique helps mitigate inconsistencies and facilitates smoother integration between the sub-models. Considering the potential benefits of this approach, I believe we can apply the insights from this paper to enhance our project. By implementing the relational approach and incorporating meta-models, we can tackle the complexity of managing functional and non-functional requirements, ensuring a more seamless integration process and reducing the likelihood of inconsistencies. This, in turn, can lead to a more efficient and robust system for our project, ultimately enhancing its overall quality and performance.

In [Andrade et al., 2002], a three-layered architectural approach is proposed, emphasizing a stringent separation between computation, coordination, and configuration, referred to as a "coordination contract." This coordination contract serves as an additional level of abstraction built on top of standard Object-Oriented Programming (OOP) constructs. we find this paper highly relevant to our project, as it aligns with similar subjects we are exploring. The concept of a coordination contract, which facilitates clear demarcation between the essential aspects of computation, coordination, and configuration, can greatly benefit our project's architecture. By incorporating the insights from this paper, we can enhance the structure and design of our project, ensuring a well-defined and efficient coordination mechanism. The use of the coordination contract can improve the maintainability and extensibility of our codebase, making it easier to manage and adapt to future changes. In conclusion, the ideas presented in [Andrade et al., 2002] resonate with the objectives and scope of our project, making it a valuable resource to

consider during the development process. Integrating the three-layered architectural approach and coordination contract principles can lead to a more robust and organized system, ultimately contributing to the success of our project.

In the context of robotics, the complexity of systems, involving hardware interactions, sensor integration, and intricate control algorithms, necessitates a disciplined approach to software design. The lack of SoC in robotics software can lead to challenges such as codebase entanglement, reduced maintainability, and hindered adaptability to evolving robotic technologies [Brooks, 1991]. The concept of SoC in robotics involves segregating concerns such as perception, control, and decision-making into distinct components. By doing so, developers can focus on individual functionalities independently, reducing the risk of unintended consequences and simplifying debugging processes [Quigley et al., 2009]. In summary, the application of SoC in the domain of robotics is a crucial aspect of software engineering that addresses the unique challenges posed by complex robotic systems. By adhering to the principles of SoC, developers can foster a more scalable, adaptable, and efficient approach to the development and maintenance of robotic software.

## 2.3 Robot Operating System

Existing methods to support Separation of Concerns in robotics encompass a range of strategies and frameworks designed to enhance the modularization, maintainability, and scalability of robotic software. One notable approach involves the utilization of the Robot Operating System (ROS) and its associated concepts. The ROS [M and al, 2009] is a widely adopted open-source middleware for robotics development. ROS provides a framework that inherently supports SoC principles by encouraging the development of modular and decoupled components. It employs a publish-subscribe communication model and a service-oriented architecture, allowing different robotic functionalities to be encapsulated in separate nodes that communicate through well-defined interfaces [Quigley et al., 2009]. ROS facilitates the separation of concerns by enabling developers to focus on specific robot functionalities, such as perception, control, and planning, within individual nodes. The modularity introduced by ROS enhances the maintainability and reusability of code, fostering a more organized and adaptable approach to robotic software engineering.

The ROS2 project aims to capitalize on the strengths of ROS 1 while addressing its shortcomings. ROS 2 utilizes DDS/RTPS as its middleware, incorporating functionalities such as discovery, serialization, and transportation. The rationale behind adopting DDS (Data Distribution Service) implementations and the RTPS wire protocol of DDS is elucidated here. In essence, DDS serves as an end-to-end middleware that offers features pertinent to ROS systems, including distributed discovery (in contrast to the centralized approach in ROS 1) and the ability to manage various Quality of Service (QoS) options for transportation. The flexibility of ROS 2 is evident in its support for multiple DDS/RTPS implementations, acknowledging that a "one size fits all" approach may not be optimal when selecting a vendor or implementation. Several factors may influence this choice, including logistical considerations like licensing, technical aspects such as platform availability, or computational footprint. Vendors may offer diverse DDS or RTPS implementations tailored to different requirements [Macenski et al., 2022a].

The ROS client library provides an API that exposes communication concepts, such as publish/subscribe, to users. In ROS 1, the implementation of these communication concepts relied on custom protocols, such as TCPROS. In contrast, ROS 2 has opted to build on top of an existing middleware solution, specifically DDS (Data Distribution Service). This strategic decision offers a significant advantage, allowing ROS 2 to leverage a mature and well-established implementation of the DDS standard. While ROS could have chosen to build on a single DDS implementation, numerous alternatives exist, each with its own set of advantages and drawbacks concerning supported platforms, programming languages, performance characteristics, memory footprint, dependencies, and licensing. Recognizing this diversity, ROS strives to support multiple DDS implementations, acknowledging the nuanced differences in their respective APIs. To manage this variability, an abstract interface has been introduced to abstract from the specifics of individual DDS APIs. This interface can be implemented for different DDS implementations, serving as a middleware interface that defines the API between the ROS client library and any specific DDS implementation [Macenski et al., 2022a].

RViz[2] (ROS Visualization) stands out as a robust 3D visualization tool crafted for ROS, empowering users to observe sensor data, robot models, and diverse information within a three-dimensional space. Its seamless integration with ROS relies on the subscription to relevant ROS topics, including

[2]http://wiki.ros.org/rviz

sensor_msgs, geometry_msgs, nav_msgs, and more, facilitating the visualization of data disseminated through these topics. User control over these subscriptions is streamlined through the left window pane, offering options to select visualizations and modify the subscribed topics. Moreover, RViz actively engages with ROS by subscribing to various topics, allowing users to manipulate these subscriptions efficiently via the left window pane. This granular control, achieved through visualization selection and subscribed topic modification, ensures precision in displaying information. Furthermore, RViz provides specific tools, such as those for setting goals or current poses, which publish messages to ROS topics, solidifying its status as a versatile and comprehensive visualization tool in the ROS ecosystem.

## 2.4 Navigation 2

In their work [Macenski et al., 2020], the authors introduced Navigation 2 as an advanced navigation solution that builds upon the successful legacy of ROS navigation. It provides a comprehensive set of services that are typically found in an operating system. These services encompass hardware abstraction, precise control over low-level devices, implementation of commonly used functionalities, seamless communication between processes via message-passing, and efficient package management. Furthermore, ROS offers a diverse range of tools and libraries to streamline tasks such as acquiring, constructing, writing, and executing code across multiple computers.The ROS runtime graph is a peer-to-peer network of processes, which can be spread across multiple machines, and they are connected using the ROS communication infrastructure. The communication in ROS can be done in various ways, such as synchronous RPC-style communication through services, asynchronous streaming of data using topics, and data storage on a Parameter Server. While ROS itself is not designed for real-time applications, it can be integrated with real-time code if needed [Dattalo, 2018]. The central objective of Navigation 2 revolves around empowering mobile robots to navigate safely and execute intricate tasks across various environments and robot kinematics. Navigation 2 surpasses mere point-to-point movement by accommodating intermediate goal points and facilitating complex actions such as object tracking. Employing a behavior tree framework, it provides a structured approach to organizing and managing novel methods and tasks for navigation. This project introduces a fully configurable, open-source navigation system encompassing three core navigation tasks: Planner,

Controller, and Recovery.

For visualizing the Navigation 2 system's communication between nodes and topics, as well as understanding their interconnections, the rqt_graph proves to be a valuable tool. The ROS computation graph visualization is facilitated by the rqt_graph graphical user interface (GUI) plugin, which has been designed with a generic structure to allow other packages to establish dependencies on it. This tool, an integral part of the Rqt suite, offers a comprehensive view of the ROS graph dynamics. With rqt_graph, users can effectively visualize the complex ROS graph of their applications, observing running nodes and communication patterns. Seamlessly integrated into the Rqt suite, this GUI plugin presents a user-friendly window into the system's architecture, displaying nodes and topics organized within their namespaces and providing an overarching system overview. As seen in Figure 2.1, which illustrates how nodes and topics connect within the Navigation 2 system, there's a lot of interconnectivity among numerous nodes. This complexity gets even more pronounced as the code expands, leading to challenges in both development and maintenance. This results in a system that's intricate and interdependent. In the upcoming discussion, we'll explore the integration of an extended Navigation 2 model, which further adds to the model's complexity and makes maintenance more challenging.

To provide a clearer representation of the complexity within the system, we opted to delve more deeply into a single node. Specifically, in Figure 2.2, we present a more detailed view of the 'bt_navigator_rclccp_node.' Furthermore, in Figure 2.3, you'll notice a similar overarching concept presented in a more general manner. Within Figure 2.2, nodes are depicted as ovaal shapes, while rectangular shapes symbolize different topics. Observing the system components—bt_navigator, controller, and planner server—it becomes evident that one topic serves data transmission, while another serves as feedback. Multiple topics are dedicated to recovery processes. This visual reveals the intricate communication interdependencies among all nodes. It's important to note that these nodes are interconnected and mutually reliant, posing potential challenges for system expansion.

Let's now take a look at a section of the code related to this node. This analysis will aid us in gaining a more effective understanding of the underlying complexity.

```
1 #include "nav2_bt_navigator/bt_navigator.hpp"
2 #include "geometry_msgs/msg/pose_stamped.hpp"
3 #include "nav2_behavior_tree/behavior_tree_engine.hpp"
```

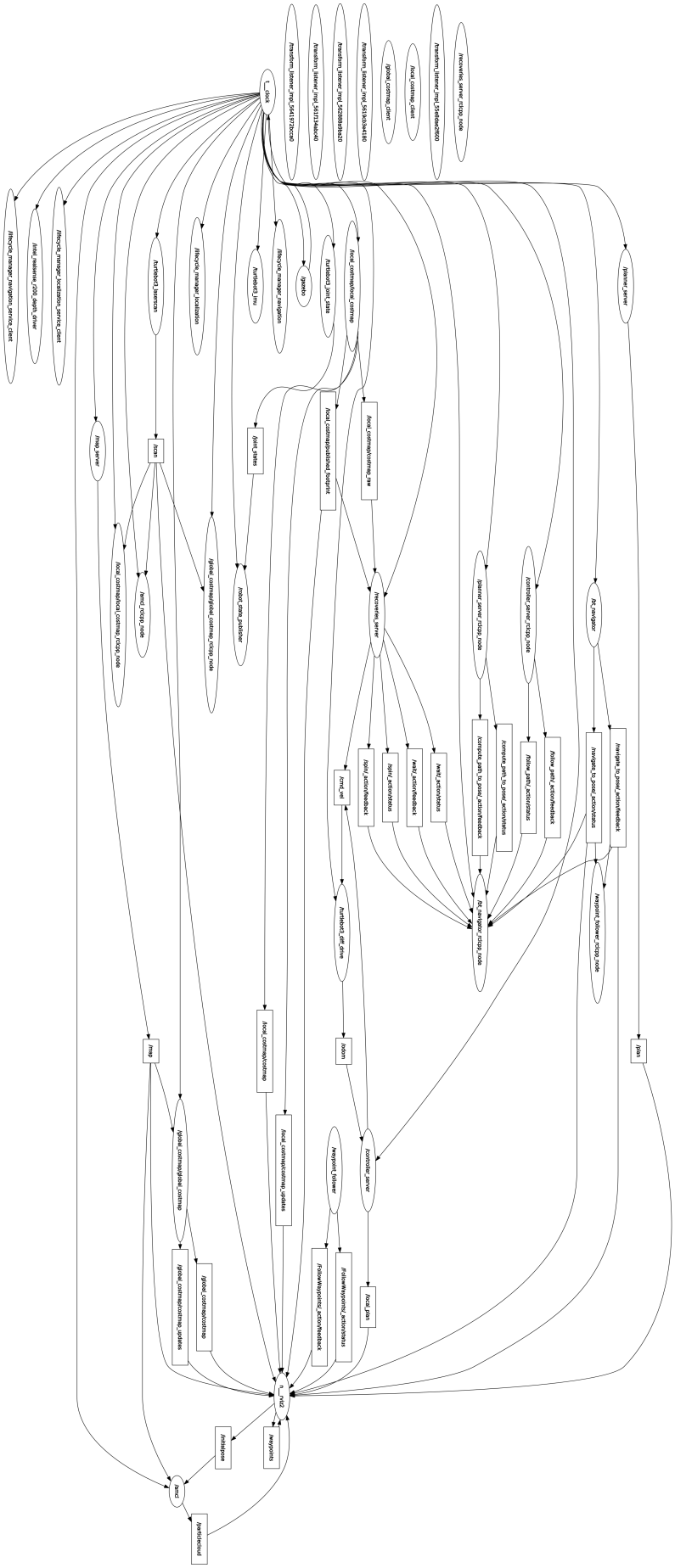

FIGURE 2.1: RQT Graph

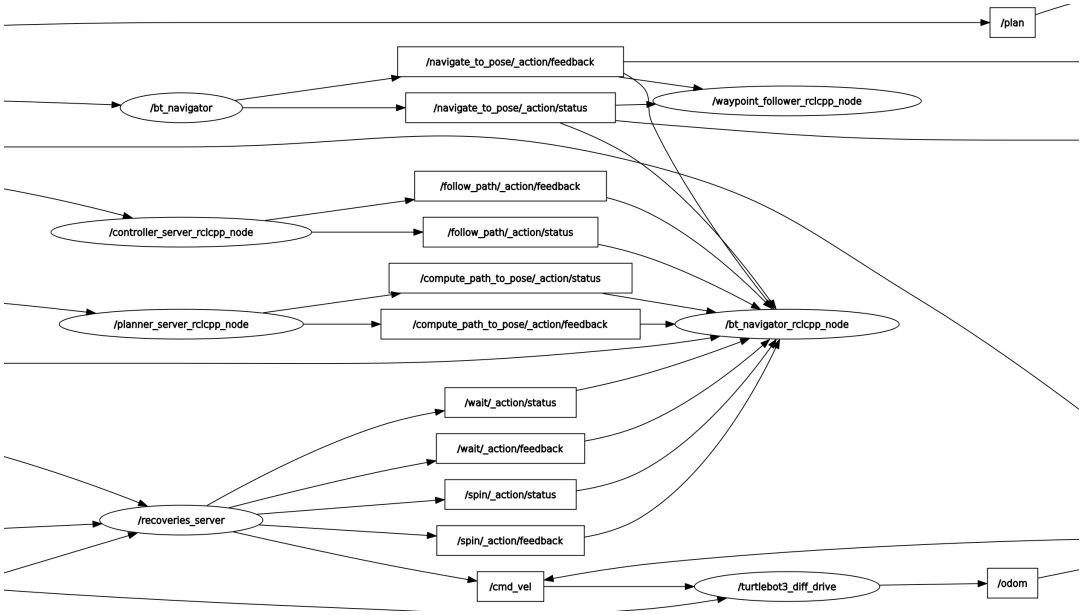


FIGURE 2.2: RQT Graph for Bt_Navigation Node

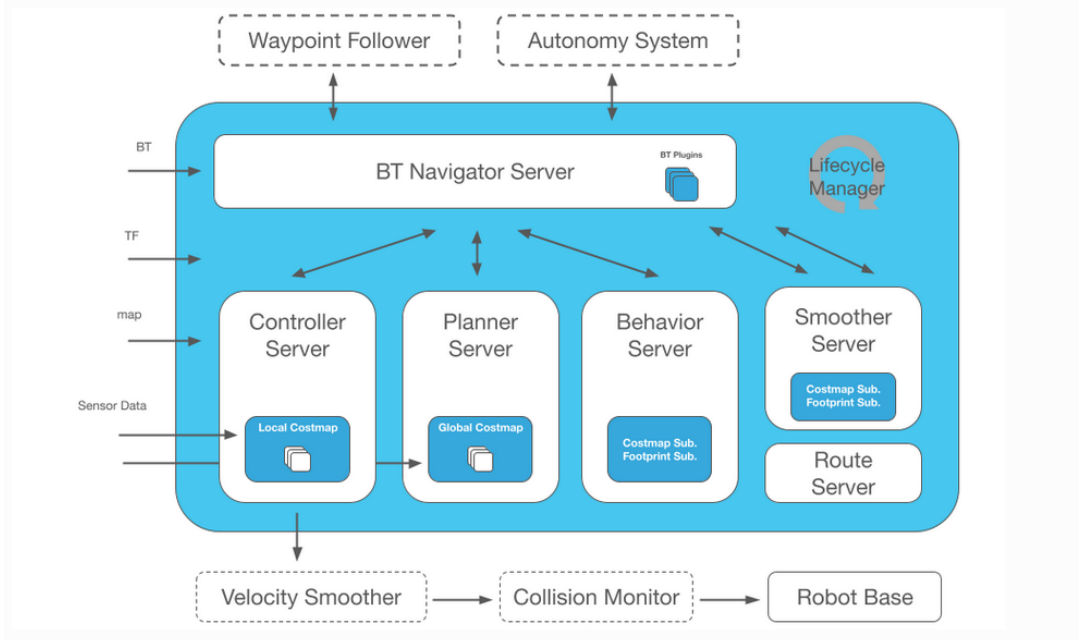


FIGURE 2.3: Behaviour Tree Navigation 2

```
#include "nav2_util/lifecycle_node.hpp"
#include "nav2_msgs/action/navigate_to_pose.hpp"
#include "nav_msgs/msg/path.hpp"
#include "nav2_util/simple_action_server.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "tf2_ros/transform_listener.h"
#include "tf2_ros/create_timer_ros.h"

#include "nav2_util/geometry_utils.hpp"
#include "nav2_util/robot_utils.hpp"
#include "nav2_behavior_tree/bt_conversions.hpp"
#include "nav2_bt_navigator/ros_topic_logger.hpp"

namespace nav2_bt_navigator
{

BtNavigator::BtNavigator()
: nav2_util::LifecycleNode("bt_navigator", "", false),
  start_time_(0)
{
  RCLCPP_INFO(get_logger(), "Creating");

  const std::vector<std::string> plugin_libs = {
    "nav2_compute_path_to_pose_action_bt_node",
    "nav2_follow_path_action_bt_node",
    "nav2_back_up_action_bt_node",
    "nav2_spin_action_bt_node",
    "nav2_wait_action_bt_node",
    "nav2_clear_costmap_service_bt_node",
    "nav2_is_stuck_condition_bt_node",
    "nav2_goal_reached_condition_bt_node",
    "nav2_initial_pose_received_condition_bt_node",
    "nav2_goal_updated_condition_bt_node",
    "nav2_reinitialize_global_localization_service_bt_node",
    "nav2_rate_controller_bt_node",
    "nav2_distance_controller_bt_node",
    "nav2_speed_controller_bt_node",
    "nav2_truncate_path_action_bt_node",
    "nav2_recovery_node_bt_node",
    "nav2_pipeline_sequence_bt_node",
    "nav2_round_robin_node_bt_node",
    "nav2_transform_available_condition_bt_node",
    "nav2_time_expired_condition_bt_node",
    "nav2_distance_traveled_condition_bt_node",
    "nav2_rotate_action_bt_node",
    "nav2_translate_action_bt_node",
```

```
50      "nav2_is_battery_low_condition_bt_node",
51      "nav2_goal_updater_node_bt_node",
52      "nav2_navigate_to_pose_action_bt_node",
53    };
54
55    // Declare this node's parameters
56    declare_parameter("default_bt_xml_filename");
57    declare_parameter("plugin_lib_names", plugin_libs);
58    declare_parameter("transform_tolerance", rclcpp::
       ParameterValue(0.1));
59    declare_parameter("global_frame", std::string("map"));
60    declare_parameter("robot_base_frame", std::string("base_link")
       );
61    declare_parameter("odom_topic", std::string("odom"));
62    declare_parameter("enable_groot_monitoring", true);
63    declare_parameter("groot_zmq_publisher_port", 1666);
64    declare_parameter("groot_zmq_server_port", 1667);
65  }
66
67  BtNavigator::~BtNavigator()
68  {
69    RCLCPP_INFO(get_logger(), "Destroying");
70  }
71
72  nav2_util::CallbackReturn
73  BtNavigator::on_configure(const rclcpp_lifecycle::State & /*
       state*/){...}
74  bool
75  BtNavigator::loadBehaviorTree(const std::string &
       bt_xml_filename) {...}
76  nav2_util::CallbackReturn
77  BtNavigator::on_activate(const rclcpp_lifecycle::State & /*state
       */) {...}
78  nav2_util::CallbackReturn
79  BtNavigator::on_deactivate(const rclcpp_lifecycle::State & /*
       state*/){...}
80  nav2_util::CallbackReturn
81  BtNavigator::on_cleanup(const rclcpp_lifecycle::State & /*state
       */){...}
82  nav2_util::CallbackReturn
83  BtNavigator::on_shutdown(const rclcpp_lifecycle::State & /*state
       */) {...}
84  void
85  BtNavigator::navigateToPose() {...}
86  void
87  BtNavigator::initializeGoalPose() {...}
```

```
88 void
89 BtNavigator::onGoalPoseReceived(const geometry_msgs::msg::
       PoseStamped::SharedPtr pose){...}
90 }
```

Throughout its development, the Navigation 2 project underwent an evolution. The core developers undertook a gradual expansion of the design, as depicted in Figure 2.4, with a specific emphasis on integrating a new world model for 2D navigation [Orduno, 2019]. The proposed design unfolds in several sequential phases, each aimed at enhancing various aspects of robotic navigation systems. This comprehensive approach encompasses diverse levels of navigation, catering to unstructured scenarios devoid of specific rules or reference paths, as well as structured navigation governed by predefined rules. Moreover, the design prioritizes adaptability by accommodating different types of planners and controllers, facilitating seamless integration and flexibility.

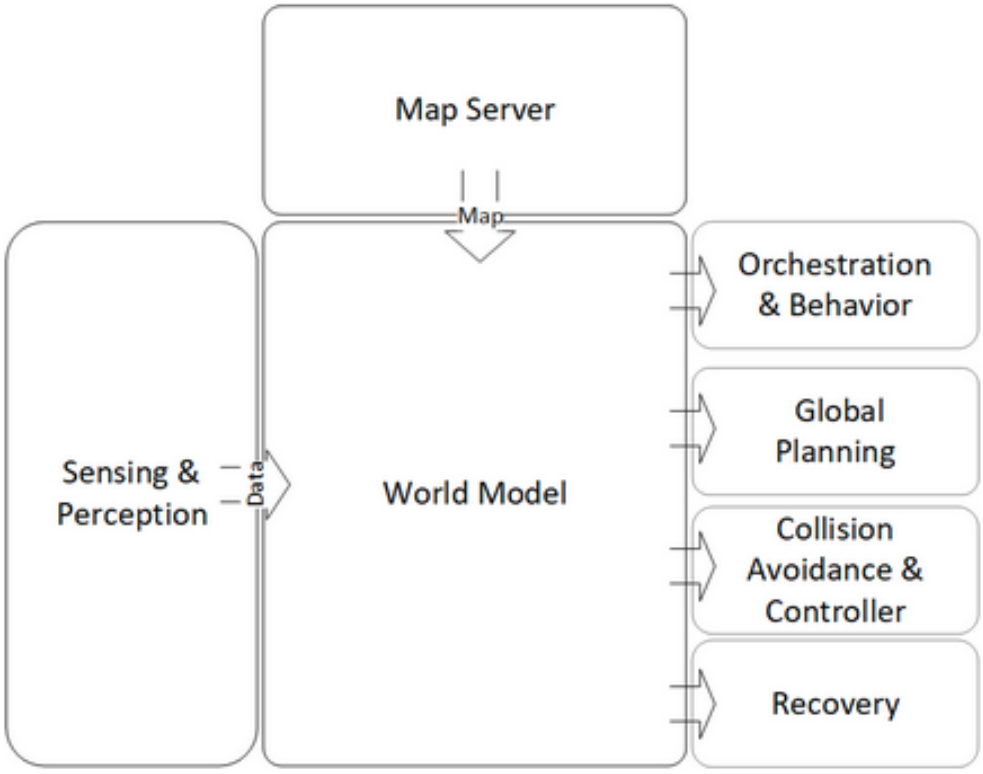


FIGURE 2.4: Current World Model

During the initial phase, a pivotal proposal emerged: the separation of the

world model from the clients, leading to the creation of distinct nodes. Subsequently, in the second phase, a groundbreaking step was taken as new modules were introduced, accompanied by the migration of the existing costmap-based world model. This phase's primary objective was to untangle the core representation from client specifics, achieved through the implementation of plugins. As the development unfolded, subsequent phases witnessed further expansion. Beyond grid-based maps, additional map formats were introduced, accompanied by the integration of advanced perception pipelines. The framework was further bolstered with support for multiple internal representations, enhancing versatility and adaptability. Culminating this evolution, the new design emerged, seamlessly depicted in Figure 2.5. In summary, their evaluation led them to conclude that the new design not only fell short of its intended objectives but also introduced unnecessary complexity. Consequently, they made the strategic decision to adhere to the current design [Orduno, 2019].

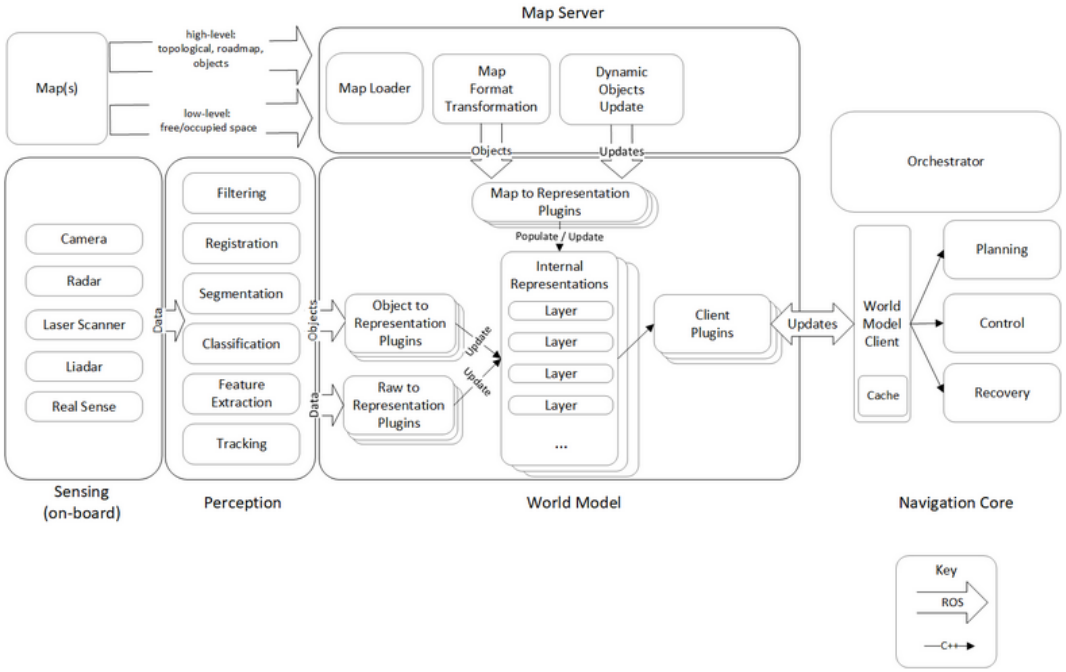


FIGURE 2.5: New World Model

## 2.5 Code Extension

Developers confront significant challenges in creating a software system that facilitates seamless code extension and modification. Their objective is to establish a software model that is both flexible and adaptable. Nonetheless,

as exemplified by the navigation system, the introduction of additional features during the software development phase often leads to intricacies arising from numerous dependencies and intercommunication among different system components. This, in turn, results in the software system growing in size, complexity, and potentially disorder, with segments of code that prove difficult to optimize[ Orduno, 2019].

Moreover, when we see interactions between different modules, it can create more problems and complexities. To handle these challenges effectively, we should break the project into smaller, self-contained tasks. This way, we can deal with specific issues more efficiently and avoid getting tangled up in complex dependencies.

I believe this approach not only streamlines our development journey but also enhances our system's maintainability and fosters a clearer comprehension of its architectural framework. Moreover, it paves the way for improved testing, debugging, and optimization of each task in isolation, culminating in an overall software solution that is more robust and efficient. Breaking down our project into autonomous tasks empowers us to navigate complexities with greater efficiency, crafting a well-organized and easily manageable software system poised for seamless adaptation to future modifications and expansions.[Chaychi, Zampunieris, and Reis, 2023].

## 2.6 Code Reusability

The concept of software reusability offers significant advantages to developers, as it allows them to utilize existing pieces of a software system to create new applications. This approach considerably reduces the time and effort required compared to building a complete software system from scratch [9]. To achieve a highly reusable system, it is essential to design modules that are independent of each other, ensuring that they can be easily integrated and repurposed for various projects.

One of the inevitable aspects of software development is the need for maintenance at some point during the project's lifecycle. As software evolves, changes, updates, and bug fixes become necessary to ensure its continued functionality and reliability. Effective maintenance practices play a crucial role in keeping the software system running smoothly and meeting user needs over time. By emphasizing reusability and prioritizing the independence of modules, developers can not only reduce development time but also

streamline the maintenance process. Reusable components can be tested, optimized, and perfected once and then applied to multiple projects, saving considerable effort and resources. Moreover, independent modules facilitate targeted updates and bug fixes, making maintenance tasks more manageable and minimizing the risk of unintended side effects. Investing in reusability and independent modules is a strategic approach that yields long-term benefits for software development projects. It empowers developers to build robust, adaptable, and maintainable software systems that can evolve with changing requirements and stay relevant in the face of technological advancements. As the software industry continues to evolve, the focus on reusability and independence will remain essential to drive innovation and ensure efficient development and maintenance practices[Chaychi, Zampunieris, and Reis, 2023].

## 2.7 Code Maintenance

Another prevalent challenge in software development is code maintenance, both during the development phase and in the years following its implementation. As projects evolve, the need for updates and bug fixes becomes inevitable. Ignoring the importance of code maintenance can lead to a deteriorating software system. To ensure a robust and reliable software product, it is essential to prioritize code maintenance and updates throughout the software's lifecycle.

In our project, recognizing the significance of maintenance in software development, we are prioritizing implementation strategies that emphasize the separation of concerns. By adopting this approach, we aim to create a system that facilitates straightforward maintenance and efficient bug-fixing processes.

To evaluate the effectiveness of the Proactive Engine approach in enhancing separation of concerns, we will employ software metrics to measure various aspects such as modularity, code coupling, and cohesion. The metrics will help assess the improvements achieved and provide quantitative insights into how the proactive scenario-based approach contributes to the separation of concerns in robotic software engineering.

## 2.8 Conclusion

Considering all these factors, separation of concerns emerges as a fundamental objective in the development of any software system. In this thesis, our primary focus will be on tackling the lack of separation of concerns by utilizing the proactive scenario-based approach of Proactive Engine, which empowers developers to strategically separate concerns, leading to a more modular and maintainable software system. Our primary research question, which we will address in this thesis, is: "How to improve separation of concerns in robotic software engineering? How can software metrics be used to measure the enhancement?" By incorporating proactive scenarios, developers can implement a well-structured and adaptable software architecture, which facilitates seamless updates, bug fixing, and code maintenance. The ultimate goal is to create a software system that is easy to understand, extend, and modify, enabling developers to respond promptly to evolving requirements and ensuring the longevity and success of the software product in the dynamic landscape of the software industry.

# Chapter 3

# Tools

In this section, we will introduce the tools that we utilized in our thesis to implement our model. These tools were instrumental in shaping the success of our research and enabled us to delve into the fascinating world of computer science and robotics.

## 3.1 Robot Operating System

Our implementation is based on ROS 2, specifically utilizing the Foxy version. ROS 2, the second generation of the Robot Operating System, aims to retain the strengths of ROS 1 while addressing its limitations. It utilizes DDS/RTPS as its middleware, providing discovery, serialization, and transportation capabilities. DDS, a comprehensive middleware used in critical infrastructure, aligns well with ROS systems by offering distributed discovery and control over Quality of Service (QoS) options. ROS 2 supports multiple DDS/RTPS implementations, considering factors such as licensing, platform availability, and computational footprint. This redesign of ROS 2 leverages community-driven capabilities to overcome challenges, utilizing the Data Distribution Service (DDS) that is widely used in critical systems. By adopting DDS, ROS 2 offers enhanced security, support for embedded and real-time applications, seamless communication between multiple robots, and the ability to operate in challenging networking environments[Macenski et al., 2022b]. As previously mentioned, ROS is accessible in several programming languages, including Python, C++, and Lisp. Experimental libraries also exist in Java and Lua. For our specific implementation, we have chosen to utilize C++.

Navigation 2 is a professionally supported project that serves as the successor to the ROS Navigation Stack. It aims to enable mobile robots to safely navigate and accomplish complex tasks across various environments and

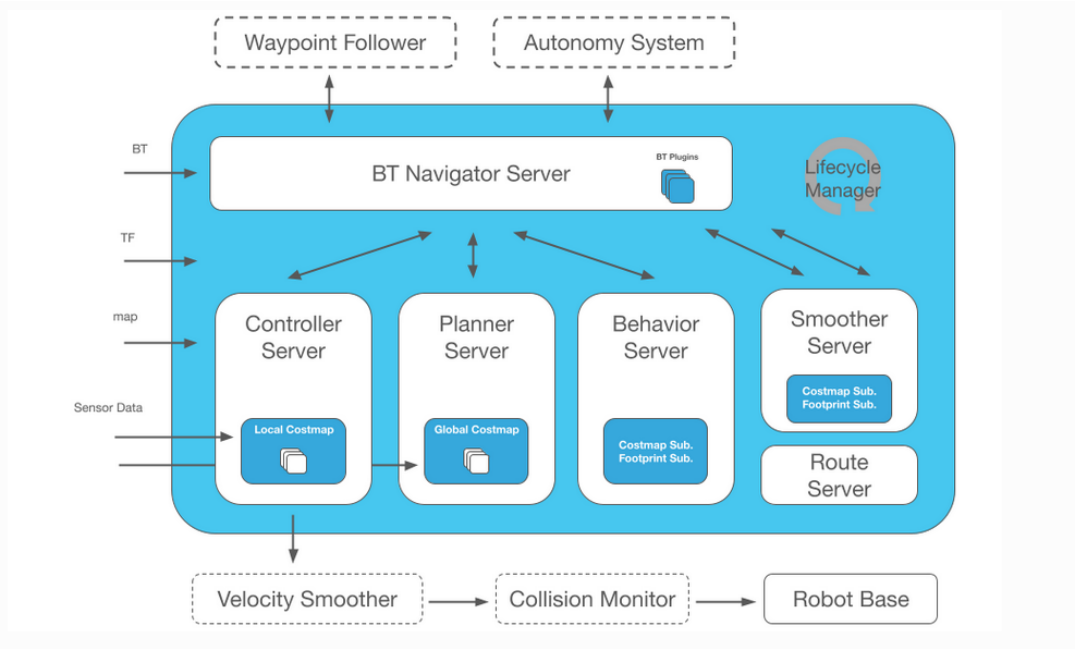


FIGURE 3.1: Navigation 2 Behaviour Tree [Macenski et al., 2020]

robot kinematics. Beyond simple point-to-point movement, Navigation 2 allows for intermediate goal point and supports diverse tasks like object following. It encompasses perception, planning, control, localization, visualization, and other essential components to build highly reliable autonomous systems. The core of navigation tasks lies in planners and controllers, which determine the robot's path. Recoveries are employed to handle challenging situations and ensure fault tolerance. Smoothers can be utilized to enhance the quality of planned paths. These highly reliable autonomous systems perform tasks such as environmental modeling based on sensor data, dynamic path planning, obstacle avoidance, and objects, and the organization of higher-level robot behaviors [Macenski et al., 2020].

## 3.2 Proactive Engine

The idea of proactive computing was first put forth by [Tennenhouse, 2000]. In proactive computing, the human user no longer occupies the central role in the interaction between humans and computers. Instead, the user serves as a supervisor who monitors the actions performed by a proactive system. A proactive system operates without depending on explicit user input, and is capable of taking actions on its own initiative [Tennenhouse, 2000]. It can respond to various events and even interpret the absence of user input in a

proactive manner. To achieve this, proactive systems must possess contextual awareness, extract pertinent information from it, and then respond accordingly to their tasks [Shirnin, Reis, and Zampunieris, 2013]. The concept of proactive computing led to the creation of a rule-based Proactive Engine (PE) by Professor Zampunieris and his team. This engine was employed in numerous projects at the University of Luxembourg, particularly in the areas of robotic[Dias, Reis, and Zampunieris, 2012], E-Learning[Dobrican, Reis, and Zampunieris, 2013], cognitive science[Shirnin, 2014], and eHealth [Dobrican and Zampunieris, 2016].

### 3.2.1 Rules

The Proactive Engine is essentially a rule-based system that integrates both object-oriented principles and rule-based systems. It includes a rule engine, database, and set of rules, which allows it to be attached directly to other systems or connected via a shared database[Chaychi, Zampunieris, and Reis, 2023]. The system is designed to be proactive, using predefined rules to processing data and this feature allows the proactive engine to be highly responsive and proactive, preventing potential issues and improving efficiency.

The rule engine executes the proactive rules in iterations and consists of two First-In-First-Out (FIFO) queues: the currentQueue and the nextQueue. The currentQueue contains rules that need to be executed in the current iteration, while the nextQueue contains rules generated during the current iteration. At the end of each iteration, the rules from the nextQueue are added to the currentQueue, and the nextQueue is emptied. This ensures that all rules are executed and the system remains efficient and optimized[Neyens, 2019]. The behavior of the rules-running system is influenced by two parameters: F, the time frequency of activation periods, and N, the maximum number of rules to be run during an activation period. These parameters are set by the system manager. The activation of the rules-running system is triggered by parameter F. If the system is already activated, it will continue with its current activation. Once activated, the system executes the first N rules in the FIFO list (if available) in order of their ranks, using the algorithm shown at the end of this section. Once a rule is executed, it is removed from the system. If a rule needs to remain active in the system for a longer period, it must clone itself to be included in the next activation of the rules-running system[Zampunieris, 2006b]. A rule can consist of any number of input parameters and comprises five distinct execution steps, each playing a unique role in the execution process[Zampunieris, 2006a].

### Data Acquisition

Data acquisition is the first step performed when a rule is run, allowing it to obtain information from the system for use in other parts of the rule. The context manager of the proactive engine provides this data, which can be obtained from various sources such as sensors or a database. The acquired data is stored in local variables, which are read-only and cannot be modified by the rule. However, these variables can be used as references to access and modify values in the system database. Once the rule is executed, the variables are discarded.

### Activation Guards

After the data acquisition part, an activation guard is performed. This guard is composed of a set of AND-connected tests on the local variables. If the activation guard is evaluated positively, the conditions and actions parts of the rule are executed. If not, these parts are ignored, but the rules generation part is still performed. A local Boolean variable called 'activated' is automatically defined and its value is set according to the result of the guard evaluation. If the activation guard evaluates to true, the 'activated' variable of the rule will also be set to true.

### Conditions

The conditions part comprises a series of tests on local variables that are connected by AND statements. These tests determine whether the subsequent actions part will be executed or not. The syntax and semantics of the conditions tests are equivalent to those of activation guards.

### Actions

The actions part, which is the fourth component, consists of a sequential list of instructions. These instructions will only be executed if all the tests in the conditions part evaluate to true.

### Rule Generation

The fifth and final component in our software model is the rules generation phase, which occurs at the end of the process. During this phase, rules have the capability to generate additional rules that will be executed in subsequent iterations. This dynamic rule generation allows for a flexible and adaptive

system, enabling the inclusion of new rules as needed. Furthermore, the rules generation component can even load itself if the logic of the system requires it, providing a self-modifying capability to the software model. This ensures that the system can continuously evolve and respond to changing requirements or conditions, enhancing its overall effectiveness and adaptability.

By utilizing this mechanism, one can develop long-lasting rules that execute actions over an extended period.The primary algorithm for executing a rule can be outlined as follows figure (3.2). In order to enhance clarity, the algorithm is presented in pseudocode format, without including low-level details:

```
1.  repeat for each data acquisition request DA
  a.  perform DA
  b.  if error then
      raise exception on system manager console and go to step 7
      else
      create new local variable and initialize it with the
          result of DA
2.  create new local Boolean variable "activated" initialized to
    false
3.  repeat for each activation guard test AG
  a. evaluate AG
  b. if result == false then go to step 6
     else if AG == last activation guard test
     then activated = true
4.  repeat for each conditions test C
  a. evaluate C
  b. if result == false then go to step 6
5.  repeat for each action instruction A
  a. perform A
  b. if error then raise exception on system manager console and
       go to step 7
6.  repeat for each rule generation R
  a. perform R
  b. insert newly generated rule as the last rule of the system
7.  delete all local variables
8.  discard rule from the system
```

FIGURE 3.2: The algorithm to run a rule [Zampunieris, 2006a]

### 3.2.2 Scenarios

The rules mentioned earlier can be categorized into scenarios, which are sets of rules executed in response to specific events. Essentially, a scenario combines rules that are essential for reacting to or taking proactive action in particular situations [Shirnin, Zampunieris, and Reis, 2012]. There are two types

of Proactive Scenarios that categorize them: Meta Scenarios and Target Scenarios.

### Meta Scenarios

Meta Scenarios are designed to provide the system with perception-centered features. Their main objective is to identify and capture events of interest and subsequently take the appropriate actions. To activate a specific Meta Scenario that aligns with the user's current activity situation, the system needs to be aware of the current state of the database. While Target Scenarios do not possess the capability to detect changes in the database, this role is fulfilled by Meta Scenarios. These Meta Scenarios operate as context-aware, continuous, and ongoing rules that never cease. Once a Meta Scenario detects a relevant event within the database, it triggers the corresponding Target Scenarios, which then execute the predefined actions. Essentially, the Meta Scenario delegates the specific task to the appropriate scenarios. Implementing Meta Scenarios typically involves integrating them into the system environment of the Learning Management Systems, enabling interactions between the user, the Proactive Engine, and the database.Meta Scenarios primarily focus on internal actions but have an effect on the user's external environment [Shirnin, Zampunieris, and Reis, 2012].

In our thesis implementation, we have incorporated various meta scenarios, including strategy, planner, controller, and decision-making meta scenarios. These meta scenarios play a crucial role in the system's operation. Based on the specific meta scenario being utilized, the system activates the corresponding set of rules. These rules govern the behavior and actions of the system, ensuring that it operates within the desired constraints and guidelines.

### Target Scenarios

The primary objective of target scenarios is to provide multiple target responses for each event or non-event detected by Meta Scenarios. These scenarios can be seen as the hands of the Proactive Engine, responsible for executing specific predefined actions such as notifications, reminders, problem prevention, and user guidance. Unlike Meta Scenarios, which are continuous rules, Target Scenarios are designed to complete their individual tasks and then become dismissed. This distinction allows for memory optimization in the system. Target Scenarios have their own areas of application, including

the system administrator environment. When creating new scenarios and rules, the focus is on maximizing the accuracy of the Proactive Engine's actions and effectively responding to users' needs by considering their cognitive aspects such as intentions, objectives, and actions [Shirnin, Zampunieris, and Reis, 2012].

In our thesis implementation, we have incorporated several target scenarios, each serving a specific task within the system. These scenarios include the strategy, planner, controller scenarios. The strategy scenario encompasses multiple strategies designed for the navigation system. It provides different approaches to achieve specific objectives and optimize the robot's movement. The planner scenario focuses on path planning for navigation. It generates command recommendations and determines the optimal route for the robot to follow based on the provided inputs and constraints. The controller scenario is responsible for controlling the robot's movements and ensuring obstacle avoidance. It implements rules and algorithms to guide the robot's actions and ensure safe and efficient navigation.

### 3.2.3 Database

In the Proactive Engine, there are two distinct types of data that are stored in a MySQL database. The first type is utilized for storing the system's state and enabling communication with external systems that are connected to the Proactive Engine. This data encompasses all the necessary information to recover the last state of the Proactive Engine in the event of a crash.The second type of data in the Proactive Engine is specifically intended for exchanging information with external systems, Navigation 2, which exist beyond the boundaries of the Proactive Engine. This data includes historical information collected from sensors, as well as the results generated by executed rules.

## 3.3 Simulation and Visualization

### 3.3.1 Gazebo

Gazebo[1] is a popular open-source 3D robotics simulator. It allows developers to create and test applications for physical robots without relying on the actual hardware, resulting in significant cost and time savings. The simulator seamlessly integrates with the Open Dynamics Engine (ODE), a robust

[1]https://classic.gazebosim.org./

physics engine written in C/C++, and utilizes OpenGL, a versatile cross-platform API for rendering 2D and 3D vector graphics. Gazebo also provides built-in support for sensor simulation and actuator control, further enhancing its capabilities[2].

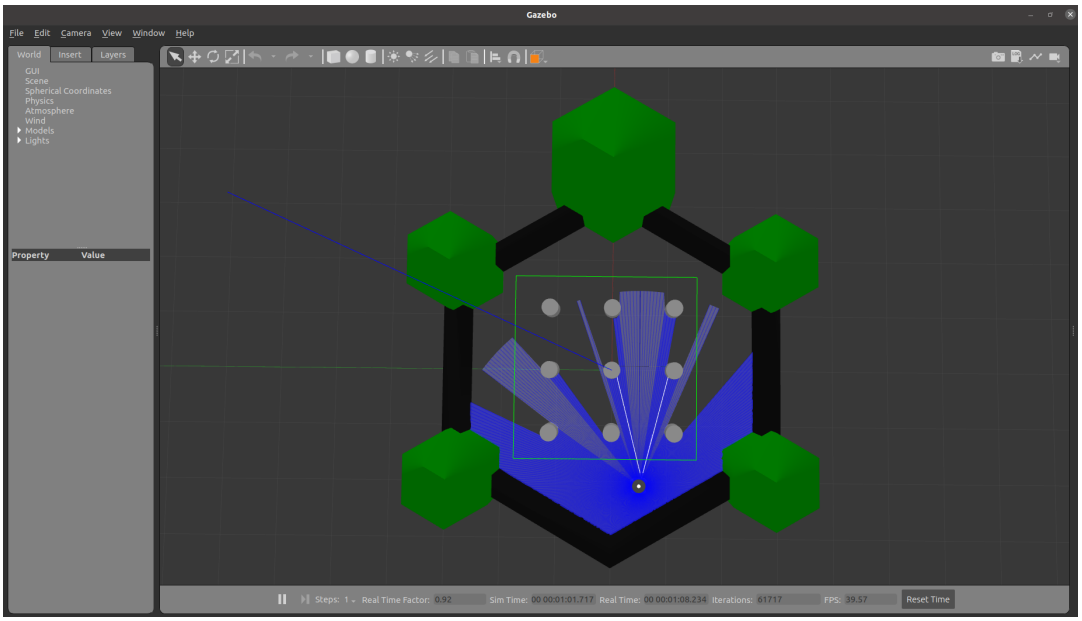


FIGURE 3.3: Gazebo Simulation

Our implementation is based on Gazebo 11, as depicted in Figure (3.3). We utilize the TurtleBot3 Simulation Package, which represents a new generation mobile robot characterized by its modularity, compactness, and customizability. The primary objective of TurtleBot3 is to significantly decrease the platform's size and cost while preserving its capability, functionality, and quality. Various optional parts, including chassis, computers, and sensors, are accessible, enabling extensive customization of the TurtleBot3 platform[3].

### 3.3.2 Rviz

Visualizing and logging sensor information plays a crucial role in the development and debugging of controllers. Rviz stands as a robust tool for robot visualization, offering a user-friendly graphical interface to visualize sensor data, robot models, and environment maps. These features prove invaluable for the development and debugging processes of the robot controllers[4]. To begin, it is necessary to launch Rviz while the robot simulation is running. Initially, we start with an empty world, to which we gradually add the robot

[2]https://en.wikipedia.org/wiki/Gazebo_simulator
[3]http://wiki.ros.org /Turtlebot3_simulations
[4]http://wiki.ros.org/rviz

model, laser scan, camera, and other required elements. Once the setup is complete, we configure the environment and save our world. Subsequently, this configured world can be reused for future sessions, eliminating the need for repetitive setup. Figure (3.4) showcases an example of your Rviz world.

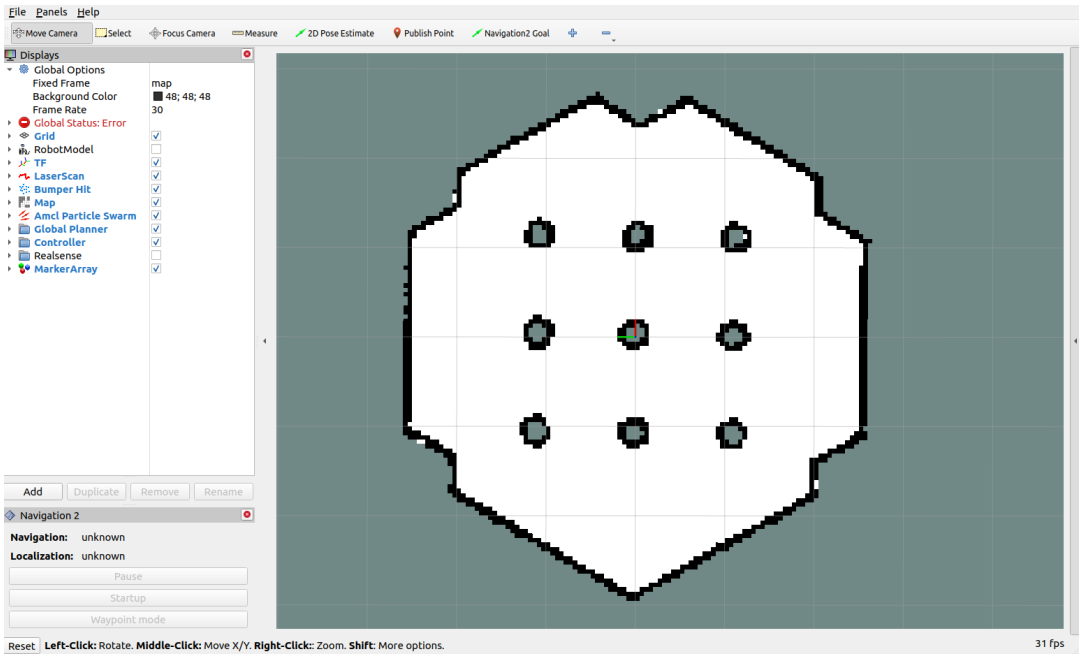


FIGURE 3.4: Rviz Visualization

Firstly, we need to establish the "2D Pose Estimate" based on the current position of the robot in the simulation or real world. This allows us to accurately initialize the robot's starting location. Next, we can set a goal point using "Navigation2 Goal". As the robot attempts to reach the initial goal, we retain the flexibility to set a new goal point if necessary. This dynamic adjustment of the goal enables the robot to adapt its path and continue moving towards the new target within the environment.

# Chapter 4

# Proposed Model

## 4.1 Introduction

In this thesis, we address the issue of insufficient separation of concerns in robotic software systems. Throughout the history of software engineering research, one of the primary challenges has been focused on enhancing modifiability, reusability, and maintainability. Our primary emphasis lies in the navigation system, utilizing a Proactive Engine to establish an enhanced navigation system that incorporates a clear separation of concerns. By incorporating a Proactive Engine into the navigation system, we aim to overcome the limitations of traditional approaches. This Proactive Engine enables a more refined separation of concerns, allowing for better modularity and encapsulation of functionality. Additionally, in our research, we utilize the ROS framework to create the robot environment. The ROS system provides a comprehensive and flexible platform for developing and managing robotic software systems. This chapter is dedicated to providing a comprehensive overview of the model. It serves as a flexible framework that can be effortlessly extended for implementing other robot systems. Furthermore, we will delve into the details of the implementation we employed for the purpose of comparing it with the Navigation 2 project, which will be elucidated in the following chapter.

## 4.2 System Architecture

In our proposed software model for navigation, we prioritize the separation of concerns to enhance the overall system architecture. By structuring the navigation system into distinct scenarios, we enable developers to focus on individual objectives and address them independently. This modular approach promotes code reusability, as each scenario can be reused in

different contexts without extensive modifications. Furthermore, it simplifies maintenance tasks by allowing developers to isolate and update specific components without affecting the entire system. To implement our software model, we utilize the Proactive Engine, a rule-based proactive system that combines object-oriented principles with rule-based systems. The Proactive Engine serves as the core engine responsible for executing the scenarios and coordinating their interactions. By running scenarios in parallel, the Proactive Engine enables efficient and independent operation without the need for explicit knowledge or communication between the scenarios. To ensure a seamless integration of the navigation system with ROS, we suggest the adoption of a database as a communication medium, as illustrated in Figure (4.1). The database serves as a repository for storing data that can be accessed by both ROS and the Proactive Engine. Within this framework, ROS components, including subscribers and publishers, interact with the database to read and write relevant data. Simultaneously, the Proactive Engine retrieves essential information from the database to make well-informed decisions and generate appropriate commands for the robot. Additionally, the ROS component retrieves the recommended command from the database, written by the Proactive Engine, in order to execute it accordingly. In the upcoming chapter, we will delve into a detailed illustration of our model.

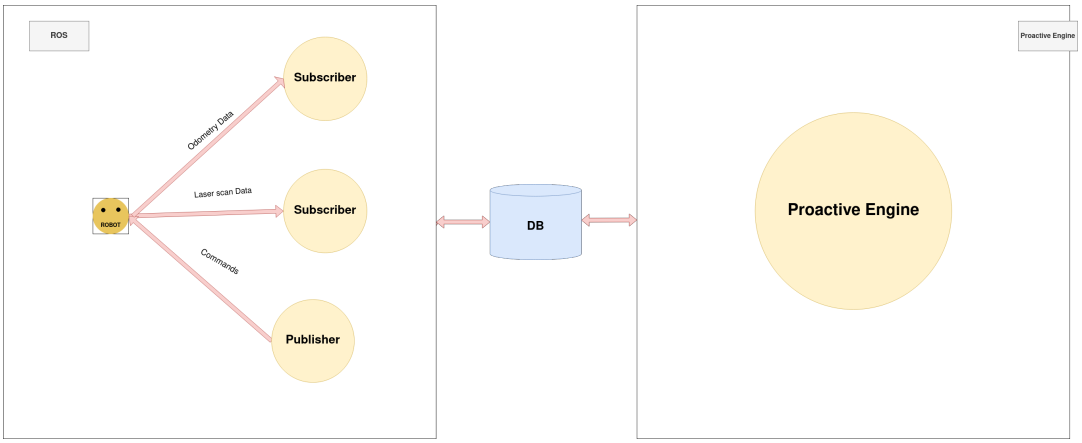


FIGURE 4.1: Our Model Architecture

## 4.3 Database

As mentioned before to facilitate the integration of the navigation system with ROS, we utilize a shared database as a central repository. In this implementation, we have opted for the use of MySQL as our database system. The

shared database acts as a communication bridge between ROS and the Proactive Engine, ensuring smooth and efficient data exchange between the two components. Through the database, ROS components such as subscribers and publishers can read and write relevant data. This allows for seamless interaction with the navigation system, enabling the retrieval of sensor data, updating of status information, and issuing commands for robot control. Simultaneously, the Proactive Engine leverages the shared database to access the necessary information required for strategy scenarios. By retrieving and analyzing data from the database, the Proactive Engine can make informed decisions based on sensor inputs, system status, and other relevant information. This facilitates the generation of appropriate commands to steer the robot effectively. The utilization of MySQL as the database system ensures reliable and efficient data storage and retrieval, contributing to the seamless integration of the navigation system with ROS. This integration promotes effective coordination between the two components, enabling robust navigation and decision-making processes within the robotic environment.

## 4.4 Proactive Engine

To implement our software model, we rely on the Proactive Engine, a rule-based proactive system that seamlessly integrates object-oriented principles with rule-based systems. The Proactive Engine consists of multiple scenarios, which can be executed in parallel, allowing for efficient and independent operation. Each scenario operates independently, without the need for explicit knowledge or communication between them. This decentralized approach simplifies the system's design and promotes scalability.

Now, let's delve into our second research question: "Is the proactive computing paradigm, implemented through a rule-based proactive engine, a suitable coding approach for addressing our initial research question? This question revolves around improving the separation of concerns in robotic software engineering. To explore this further, let's examine our proposed model in the context of the second research question.

Our design consists of two types of scenarios: meta scenarios and scenarios. Within our system, we have several scenarios, including Strategy, Planners, Controllers, and Decision Making. Each scenario has its own corresponding meta scenario as well. These scenarios are responsible for handling various aspects of the system's behavior and are designed to accomplish specific objectives and tasks. Each objective is assigned to a separate scenario,

enabling us to focus on individual scenarios, facilitating better understanding, modification, and system maintenance. Furthermore, this segregation allows us to reuse and combine different scenarios, resulting in the creation of more intricate and adaptable behaviors. Meta scenarios, on the other hand, are responsible for activating the rules within the scenarios.

### Strategy Scenario

The proposed software model for navigation consists of the Strategy scenario, which receives data from the database and utilizes the required information from it. Additionally, the Strategy scenario also receives data from the system, enabling runtime changes in the system. By incorporating multiple planned strategies, the Strategy scenario offers various options to control the robot's behavior, allowing for different behaviors to be achieved without modifying any code within the system. This flexibility enables the application of different strategies during runtime without the need to relaunch the system. To facilitate the selection of a planned strategy based on conditions and rules from the environment or user input, a meta strategy scenario is employed. This meta scenario is responsible for choosing the appropriate strategy and storing it in local storage. Other scenarios, such as the Controller, Planner, and other relevant components, can then access the selected strategy. These scenarios can then be activated based on the planned strategy, effectively coordinating the robot's actions.

```
1     Algorithm: StrategyScenario
2
3 Input:
4   Database: Object representing the database
5   SystemData: Data from the system
6   UserInput: Input provided by the user
7   MetaStrategyScenario: Object representing the meta scenario
     for strategy selection
8   ChosenStrategy: Selected strategy
9
10 1. Initialize the Database for storing and retrieving relevant
    information.
11 2. Initialize SystemData and UserInput for obtaining contextual
    information.
12 3. Create StrategyScenario within the Proactive Engine.
13 4. Logically connect StrategyScenario with its corresponding
    meta scenario (MetaStrategyScenario).
14    a. Set MetaStrategyScenario to activate rules within
    StrategyScenario.
```

```
5. StrategyScenario Operation:
   a. Receive data from the database and the system (Database,
   SystemData).
   b. Offer multiple strategies to control the robots
   behavior.
   c. Enable runtime changes in the system (SystemData).
   d. Allow different behaviors without modifying system code.
   e. Facilitate the selection of a planned strategy based on
   conditions and rules.
   f. Store the appropriate strategy in local storage (Database)
   .
6. End Algorithm
```

**Planner Scenario**

The Planner scenario plays a crucial role in our system by managing path planning and navigation. Although there are several algorithms available for implementation, we choose to activate only one Planner scenario at a time, each representing a different algorithm. To coordinate this process, we have a meta-planner scenario that retrieves the planned strategy from local storage. It then triggers the corresponding scenario, activating the algorithm that aligns with the planned strategy obtained from the strategy scenarios. This approach ensures efficient and streamlined path planning based on the specific needs and objectives of the system. Within our planner scenario, we have implemented a feature that allows the robot to modify its path and divert towards a new direction based on certain criteria. This additional functionality ensures that the robot can dynamically adapt its navigation strategy as needed. By periodically evaluating the relevant criteria, the planner scenario can determine if a path adjustment or diversion is required. This adaptive behavior enables the robot to effectively navigate and overcome obstacles or changing conditions. Integrating this capability into the planner scenario enhances the overall efficiency and autonomy of the robot, allowing it to intelligently respond to different situations and optimize its strategy.

```
    Algorithm: PlannerScenario

Input:
  MetaPlannerScenario: Object representing the meta scenario for
    planner coordination
  LocalStorage: Storage for accessing planned strategy
   information
  PathPlanningAlgorithm: Selected path planning algorithm

```

```
1. Initialize MetaPlannerScenario for coordinating the Planner
   Scenario.
2. Initialize LocalStorage for accessing planned strategy
   information.
3. Activate PathPlanningAlgorithm based on the selected
   algorithm.
   a. Set MetaPlannerScenario to retrieve planned strategy from
   LocalStorage.
   b. Activate PlannerScenario based on the planned strategy.
4. PlannerScenario Operation:
   a. Manage path planning and navigation.
   b. Activate only one Planner scenario at a time, representing
    a different algorithm.
   c. Retrieve planned strategy from LocalStorage using
   MetaPlannerScenario.
   d. Activate corresponding PathPlanningAlgorithm based on the
   planned strategy.
   e. Ensure efficient and streamlined path planning aligned
   with system needs.
   f. Periodically evaluate relevant criteria to determine if a
   path adjustment or diversion is required.
   g. Dynamically adapt the robot's navigation strategy as
   needed.
5. End Algorithm
```

### Controller Scenario

The Controller scenarios hold the responsibility of governing the movement and response of the robot in accordance with its environment. Depending on the chosen strategy, it is possible to activate multiple Controller scenarios simultaneously, leading to enhanced intelligence and improved control capabilities of the robot. Within our model, the Controller scenarios are accompanied by a meta-controller scenario. This meta-controller retrieves the planned strategy from local storage and triggers the corresponding Controller scenarios, which could be one or multiple, aligned with the expected behavior of the robot. By employing this approach, we ensure that the robot operates in a manner that is consistent with the intended objectives and desired performance, allowing for efficient adaptation and response to different situations.

```
    Algorithm: ControllerScenario

Input:
  MetaControllerScenario: Object representing the meta scenario
   for controller coordination
```

```
  LocalStorage: Storage for accessing planned strategy
   information
  RobotMovement: Object representing robot movement capabilities
  RobotResponse: Object representing robot response capabilities

1. Initialize MetaControllerScenario for coordinating the
   Controller Scenarios.
2. Initialize LocalStorage for accessing planned strategy
   information.
3. Initialize ChosenStrategy based on the selected strategy.

4. Create ControllerScenario within the Proactive Engine.
   a. Set MetaControllerScenario to retrieve planned strategy
   from LocalStorage.
   b. Activate ControllerScenario based on the planned strategy.

5. ControllerScenario Operation:
   a. Govern the movement and response of the robot in
   accordance with its environment.
   b. Activate multiple Controller scenarios simultaneously
   based on the chosen strategy.
   c. Retrieve planned strategy from LocalStorage using
   MetaControllerScenario.
   d. Activate corresponding Controller scenarios based on the
   planned strategy.
   e. Align Controller scenarios with the expected behavior of
   the robot.
   f. Operate in a manner consistent with the intended
   objectives and desired performance.
   g. Allow for efficient adaptation and response to different
   situations in the robot's environment.

6. End Algorithm
```

### Decision Making Scenario

The Decision Making (DM) meta scenario is a pivotal component within our software system. It receives data from the Controller and Planner modules in the form of recommendation commands generated by various scenarios, each with its own assigned priority. These priorities are determined during scenario implementation. By incorporating priorities into the command recommendations at their creation, we ensure that the DM scenario can seamlessly integrate new scenarios without requiring any modifications. This flexibility enables the addition of new scenarios to the system without disrupting

the decision-making process. Based on the assigned priorities, the DM scenario takes into account all the recommendation commands and makes the final decision. It then sends the selected recommendation command to the robot for execution.
The DM scenario plays a crucial role in our software system, and we have incorporated a significant feature called the feedback loop. This feedback loop enables the DM scenario to not only send the recommended command to the robot for execution but also influence the strategy scenario by sending what we refer to as a system command. Based on the newly planned strategy, the system's behavior can adapt to the current situation. This feedback loop enhances the dynamic nature of the system by allowing the DM scenario to influence the strategy scenario. As a result, the decision-making process is not limited to a one-way flow of commands but incorporates a continuous loop of information exchange. This facilitates real-time adjustments and ensures that the system can respond effectively to changing conditions.

It's important to note that the other scenarios within our system are unaware of the existence of the DM scenario. Each scenario independently generates its own decision by producing a command recommendation, which is indirectly transmitted to the DM scenario through the database. This architecture ensures autonomy and encapsulation of decision-making responsibilities while facilitating efficient communication between the different components of the system.

```
    Algorithm: DecisionMakingScenarioOperation

Input:
  ControllerRecommendations: Data from Controller modules
   containing recommendation commands
  PlannerRecommendations: Data from Planner modules containing
   recommendation commands
  PriorityAssignments: Assigned priorities for each
   recommendation command
  Database: Object representing the shared database for
   communication
  SystemCommand: Object representing the system command
   influenced by the DM scenario
  RobotExecution: Object representing the robot for command
   execution

1. Initialize ControllerRecommendations for receiving data from
   Controller modules.
2. Initialize PlannerRecommendations for receiving data from
   Planner modules.
```

```
3. Initialize PriorityAssignments for assigned priorities for
   recommendation commands.
4. Initialize Database for efficient communication between
   components.
5. Initialize SystemCommand for storing the system command
   influenced by the DM scenario.
6. Initialize RobotExecution for handling command execution by
   the robot.
7. Create DecisionMakingScenario within the Proactive Engine.

8. DecisionMakingScenario Operation:
   a. Receive recommendation commands from
   ControllerRecommendations and PlannerRecommendations.
   b. Receive priority assignments for each recommendation
   command.
   c. Incorporate priorities into command recommendations during
    their creation.
   d. Take into account all recommendation commands and make the
    final decision.
   e. Send the selected recommendation command to RobotExecution
    for execution.
   f. Store the selected recommendation command in SystemCommand
   .
   g. Implement a feedback loop:
      i. Influence the Strategy Scenario by sending a system
   command.
      ii. Adapt the system's behavior to the newly planned
   strategy.
   h. Facilitate real-time adjustments to respond effectively to
    changing conditions.

9. End Algorithm
```

## 4.5 Robot Operating System

Within the ROS framework, we seamlessly integrate essential components to enhance our robotic simulation. Gazebo, a robust robot simulator, becomes the cornerstone of our simulation environment, delivering an authentic virtual setting that closely mirrors real-world scenarios. To further amplify the user experience and interaction within this simulation, we employ RViz. This choice is not without merit, as RViz confers the dynamic capability of altering the robot's objectives in real-time, thereby imparting a remarkable flexibility to our navigation system.

Our ROS component design features a straightforward configuration, comprising two subscribers and one publisher. These components harmoniously work together to facilitate seamless data flow and efficient control. The subscribers possess distinct roles: one captures Laser scan data from the robot, with another dedicated to Odometry data. This collective effort accumulates the data in our database, positioning it for streamlined management and insightful analysis. We deliberately chose to allocate separate subscribers for different data streams, ensuring a modular approach. This architecture affords the flexibility to effortlessly introduce new data sources as needed, further bolstering the extensibility of our system. Conversely, a solitary publisher plays a pivotal role in our setup. This publisher becomes the conduit through which final commands from the database are channeled to the robot. The robot then faithfully interprets and executes these directives, effectively performing a diverse array of actions. The choice of C++ as the programming language for the ROS segment offers a strategic advantage. It empowers us with fine-grained control over the system, providing the speed and adaptability necessary for real-time operations and seamless communication within the expansive ROS ecosystem.

## 4.6 Conclusion

In this thesis, we addressed the challenge of insufficient separation of concerns in robotic software systems. Our proposed model focused on enhancing modifiability, reusability, and maintainability by incorporating a Proactive Engine into the navigation system. The model utilized modular scenarios such as Strategy, Planner, Controller, and Decision Making, each with its own responsibilities and objectives. The integration of a feedback loop allowed for dynamic adaptation and interaction between decision-making and strategy scenarios. By utilizing a shared database and the ROS framework, efficient communication and seamless integration were achieved. Overall, the proposed model provides a flexible framework for developing and managing robotic software systems, promoting modularity and adaptability.

# Chapter 5

# Implementation

## 5.1 Introduction

In this chapter, we will provide a detailed explanation of the implementation of our proposed model. In our implementation, as shown in Figure (5.1), we have established a connection between ROS and Proactive Engine through a MySQL database.This architecture allows for seamless communication between ROS and Proactive Engine, enabling the exchange of data and commands in real-time. By utilizing the database as an intermediary, we ensure reliable and efficient data transfer between the two systems.

The Proactive Engine is an implementation of a rule-based proactive system, consisting of several scenarios, each containing condition-action rules. These scenarios run in parallel, operating independently, and without awareness of each other's existence, as depicted in Figure (5.2). The detailed design of the navigation model in the Proactive Engine demonstrates that all scenarios operate concurrently and autonomously, without the need for communication between them.

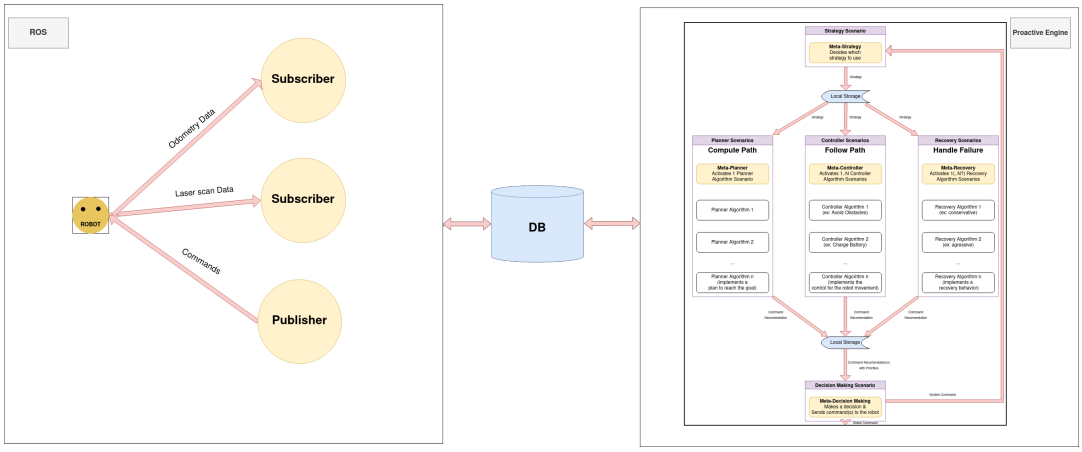


FIGURE 5.1: Architecture of the ROS and Proactive Engine implementation with a MySQL database connection.

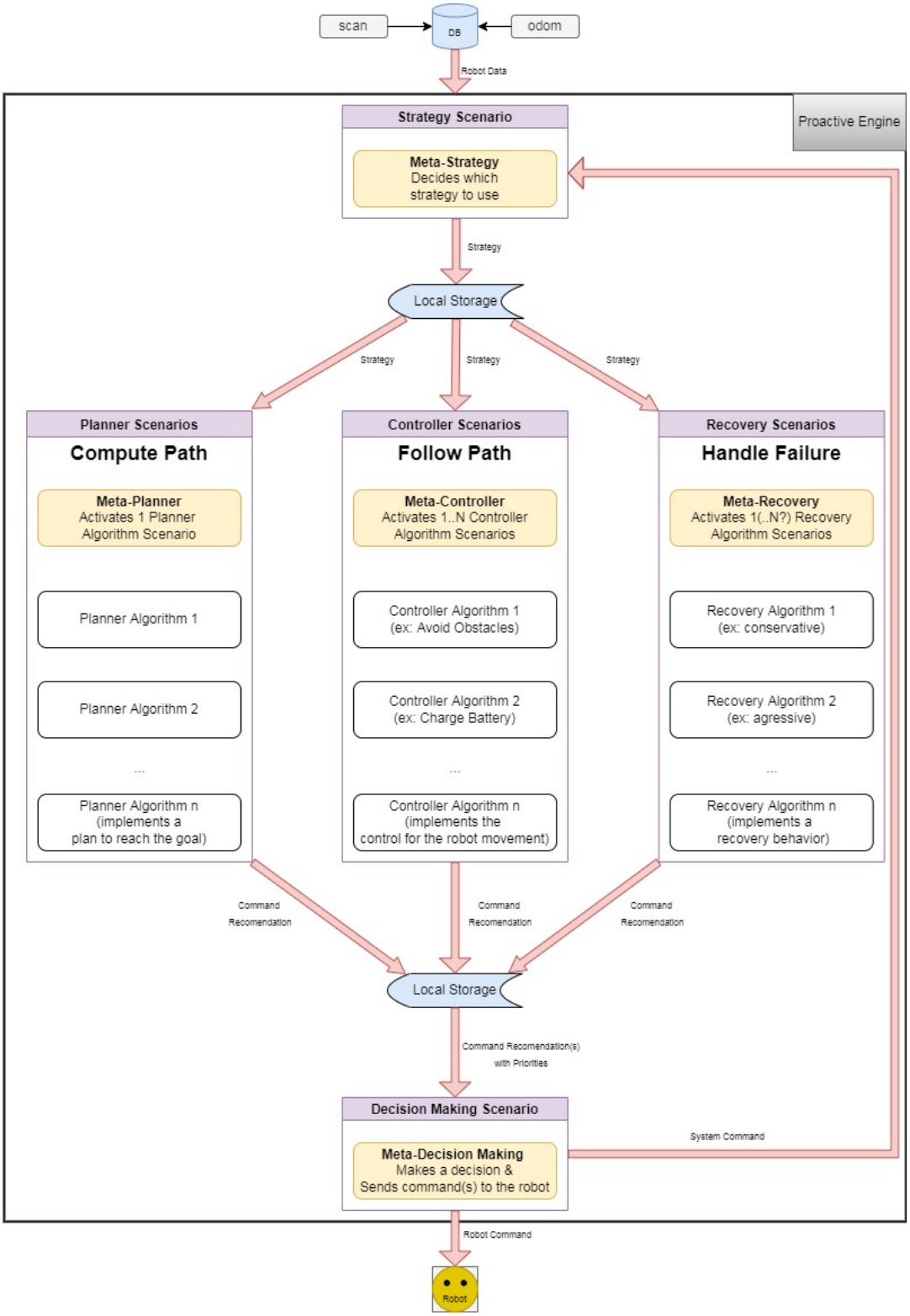


FIGURE 5.2: Navigation Model

In our ROS implementation, we utilize subscriber and publisher nodes. The subscriber nodes receive essential data from the robot and write it into the database. On the other hand, the publisher node reads data from the database and sends it to the robot for execution. This setup ensures efficient data management and seamless communication between the robot and the database, enabling the robot to perform its tasks effectively based on the received data.

Overall, this design promotes modularity and flexibility, allowing for easy integration of different components and systems within the overall architecture. It also facilitates the implementation of complex functionalities and the coordination of multiple processes in a distributed environment. Below, you will find a comprehensive overview of the detailed implementation of our system.

## 5.2 Proactive Engine

In our implementation, we have developed a navigation system consisting of several scenarios and meta-scenarios, each focusing on specific concerns. These scenarios run in parallel, working collectively to achieve our system's objectives. Let's explore how these scenarios interact and coordinate. In the Strategy scenario, the planned strategy is selected based on user or system input. Once determined, the planned strategy is saved in the local database for future access. The Controller and Planner meta-scenarios can access this local database, and within their respective meta-scenarios, they activate the corresponding rules based on the active strategy. Each scenario generates its recommended command, which is sent to the Decision Making scenario. In the Decision Making scenario, all the recommended commands from the Controller, Planner, and other scenarios are considered, along with their assigned priorities. The Decision Making scenario evaluates these inputs and makes the final decision about the command to send for the robot to act on. The selected command is then sent to the database. This synchronized interaction between the different scenarios and meta-scenarios ensures efficient coordination and seamless decision-making within our navigation system. By dividing responsibilities and utilizing a local database, our implementation achieves a modular design.

In the detailed design of the Proactive Engine, depicted in Figure (5.2), all scenarios operate in parallel and independently, with no awareness of each other's existence. This design fosters efficient communication and seamless

coordination among the various scenarios. Our implementation comprises of several essential scenarios, including Strategy, Planner, Controller, and Decision Making. Each active scenario runs in parallel, generating command recommendations based on its unique algorithm. This decentralized approach guarantees efficient and dynamic decision-making, while also preserving a modular and adaptable system architecture. In the subsequent sections, you will find a detailed explanation of the implementation of each scenario.

### Strategy

In our implementation, the Strategy scenario, as depicted in the proposed model, offers various pre-defined strategies to control the robot's behavior. These strategies include options such as going to the goal point while avoiding obstacles, going to the goal point without considering obstacles, and checking the robot's battery level to divert towards the charging station if necessary. The system can dynamically switch between these behaviors at runtime without requiring a system relaunch. For instance, if the robot's battery level is low, it will autonomously change its direction to reach the charging station, thereby altering its behavior. Afterward, when the battery has been charged, the system is able to resume the previous strategy without any intervention. Similarly, if the robot needs to consider obstacles in its path during runtime, the strategy can be adjusted accordingly. Our system is capable of choosing different scenarios based on specific conditions and rules. These conditions can be derived from the environment or user input. The feedback loop, as shown in Figure (5.2), allows the Decision Making scenario to send a command to the Strategy scenario, enabling a change in the planned strategy at runtime. The Strategy scenario stores the planned strategy in local storage, and the meta scenarios can access this data. Consequently, the corresponding scenarios from the Planner and Controller, that implement the new strategy are activated by their respective meta scenarios.

```
1  protected void actions Strategy Scenario() {
2    super.actions();
3
4    if (!MySQLOperations.isResultSetEmpty(this.sysCmdRS)) {
5      Global_Vars.logger.fine("NEW command to change strategy
   found. Reacting accordingly!");
6      try {
7        boolean firstTreated = false;
8        this.sysCmdRS.beforeFirst();
9        while (!sysCmdRS.isClosed() && sysCmdRS.next()) {
10         if (!firstTreated) {
```

```
                    applyNewSystemCommand(sysCmdRS.getString("
  command"));
            firstTreated = true;
          }
          dataNativeSystem.setCommand2RobotTreated(sysCmdRS.
  getLong("idcommand2robot"));
        }
      } catch (final SQLException e) {
        Global_Vars.logger.warning("ResultSet not empty, but '
  command' field cannot be read!");
        e.printStackTrace();
      }
    } else {
      Global_Vars.logger.fine("NEW strategy by user. Reacting
  accordingly! Code = " + userStrategyCode);
      previousStrategyCode = currentStrategyCode;
      currentStrategyCode = userStrategyCode;
    }

    dataNativeSystem.changeActiveStrategy(currentStrategyCode);
  }
```

```
  private void applyNewSystemCommand(final String newCommand) {
    if (newCommand.compareToIgnoreCase(previousSystemCommand) ==
    0) {
      return;
    } else {
      previousSystemCommand = newCommand;
    }

    switch (newCommand) {
    case CMD_RESUME:
      final String swapStrategy = previousStrategyCode;
      previousStrategyCode = currentStrategyCode;
      currentStrategyCode = swapStrategy;
      break;
    case CMD_BATTSAVE:
      previousStrategyCode = currentStrategyCode;
      currentStrategyCode = STRATEGY_CHARGE;
      break;
    default:
      Global_Vars.logger.warning("Command (CMD) not recognized!
  Ignoring ...");
      previousStrategyCode = currentStrategyCode;
      currentStrategyCode = STRATEGY_NULL;
      break;
    }
```

```
24      Global_Vars.logger.config("Strategy is now " +
        currentStrategyCode);
25    }
```

### Planner

The Planner scenario plays a crucial role in the system's implementation by computing the path based on the start and goal points, guiding the robot towards its destination. It is the only scenario that knows the goal point. Notably, there are multiple algorithms available for the Planner module, resulting in different behaviors. For example, one algorithm may prioritize turning before moving forward, while another may execute both actions simultaneously. To ensure efficient operation, the meta-Planner activates only one Planner scenario at a time, aligned with the chosen strategy. In our implementation, the "turn&move" scenario is activated, enabling the robot to efficiently reach the designated goal point. Furthermore, we have incorporated a battery level checking feature within the planner scenario. This functionality assesses whether the robot needs to modify its path and divert to a charging station for recharging. By periodically evaluating the battery level, the planner scenario determines if sufficient charge remains to complete the planned path. If not, the scenario generates a new system command to change strategy, so that the robot can go to the nearest charging station. After recharging, the robot resumes its journey towards the goal point. This adaptive behavior allows the robot to proactively address its energy needs and continue its operation effectively. Moreover, our implementation offers the flexibility to change the goal point while the robot is in motion or after recharging. Users or the system can modify the goal point as needed during navigation. This dynamic capability enhances the overall efficiency and autonomy of the robot, ensuring intelligent energy resource management and seamless navigation adjustments based on real-time requirements.

```
1   protected void actions Planner() {
2     super.actions();
3
4     if (checkpoint.sameCoordinate(robotPosition, TRESHOLD)) {
5       reachedDestination();
6     } else {
7       final double ThetaWithYaw = checkpoint.calculateAngle(
      robotPosition, robotOrientation);
8
9     }
10      selectMovement(ThetaWithYaw);
```

```
    }

    createCommandRecommendation(THIS_CMD, P_PARAM_STRING,
  THIS_PRIORITY);
}
```

```
private void selectMovement(final double ThetaWithYaw) {

    THIS_PRIORITY = PRIORITY_DEFAULT; // using the default
  priority

    if ((ThetaWithYaw < (-(Math.PI - threshold))) || (
  ThetaWithYaw > (Math.PI - threshold))) {
      THIS_CMD = "M_F";
      P_PARAM_STRING = "CMD_PARAM_M_F";
    } else if ((ThetaWithYaw > (-threshold)) && (ThetaWithYaw <
  (threshold))) {
      THIS_CMD = "M_B";
      P_PARAM_STRING = "CMD_PARAM_M_B";
    } else if ((0 < ThetaWithYaw) && (ThetaWithYaw < (Math.PI)))
    {
      THIS_CMD = "T_R_F";
      P_PARAM_STRING = "CMD_PARAM_T_R_F";
    } else if ((ThetaWithYaw < 0) && (ThetaWithYaw > -(Math.PI))
  ) {
      THIS_CMD = "T_L_F";
      P_PARAM_STRING = "CMD_PARAM_T_L_F";
    } else {
      Global_Vars.logger.severe("Angle not correct! ThetaWithYaw
  =" + ThetaWithYaw);
      return;
    }
}
```

### Controller

In the implementation of the Controller, its main task is to control the robot's movement while effectively avoiding obstacles in the environment. The Controller scenario takes charge of the robot's movement and responds to the surrounding environment. Within our implementation, the Controller includes a meta-controller scenario that retrieves the planned strategy from local storage and activates corresponding scenarios based on the expected behavior of the robot. Depending on the strategy, multiple Controller scenarios can be activated simultaneously. In our implementation, scenarios like "avoid and

move to the left" and "avoid and move to the right" are available. These chosen scenarios can be dynamically changed at runtime, depending on specific conditions and requirements. This dynamic selection of Controller scenarios ensures effective obstacle avoidance and adaptability during the robot's navigation.

```
1     protected void actions Controller() {
2    super.actions();
3
4    long cmd = 0;
5    String PARAM_STRING = null;
6
7    if (((range[0] >= 0) && (range[0] < OBJ_DISTANCE))
8        || ((range[19] >= 0) && (range[19] < OBJ_DISTANCE))
9        || ((range[16] >= 0) && (range[16] < OBJ_DISTANCE))) {
10     PARAM_STRING = ParametersData.getInstance().
   getParameterValueByName(P_PARAM_STRING_R);
11     cmd = CommandList.getInstance(getDataNativeSystem()).
   getCommandIdByName(THIS_CMD_R);
12    } else if (((range[0] >= 0) && (range[0] < this.OBJ_DISTANCE
   ))
13        || ((range[1] >= 0) && (range[1] < OBJ_DISTANCE))
14        || ((range[4] >= 0) && (range[4] < OBJ_DISTANCE))) {
15     PARAM_STRING = ParametersData.getInstance().
   getParameterValueByName(P_PARAM_STRING_L);
16     cmd = CommandList.getInstance(this.getDataNativeSystem()).
   getCommandIdByName(THIS_CMD_L);
17    } else if (((range[1] > OBJ_DISTANCE) && (range[19] >
   OBJECT_DISTANCE_LESS))
18        || (range[5] < OBJECT_DISTANCE_LESS) || (range[15] <
   OBJECT_DISTANCE_LESS)
19        || (range[7] < OBJECT_DISTANCE_LESS) || (range[13] <
   OBJECT_DISTANCE_LESS)) {
20     PARAM_STRING = ParametersData.getInstance().
   getParameterValueByName(P_PARAM_STRING_FWR);
21     cmd = CommandList.getInstance(getDataNativeSystem()).
   getCommandIdByName(THIS_CMD_FWD);
22    }
23
24    if (cmd != 0) {
25     final CommandRecommendation newCR = new
   CommandRecommendation(PRIORITY, cmd, PARAM_STRING,
26           getEngine().getIterationCount(), toString(),
   getDataNativeSystem());
27     newCR.saveDataToDB();
28    }
```

```
29  }
```

### Decision Making

Finally, we have the Decision Making(DM) scenario, responsible for reading recommendation commands and priority levels from the active Controller, Planner, and Recovery scenarios. It plays a crucial role in making the final decision based on the priorities and types of each active scenario, ultimately sending the command to the robot for execution. In this process, the decision-making scenario also dynamically influences the planned strategy by sending commands to the strategy scenario.

Let's now turn our attention to the third research question: "How can effective management of conflict handling be achieved in proactive decision-making scenarios, particularly when conflicting recommendations emerge from distinct, independently objective-based situations?" Additionally, we'll explore our fourth research question: "How can intelligent and self-managing behavior be formulated and put into action within the system, so as to tackle our third research question? This involves facilitating dynamic changes in strategy".

The feedback loop, as shown in Figure (5.2), illustrates the bidirectional flow of information between the decision-making scenario and the strategy scenario. This feedback loop enables real-time adjustments to the planned strategy, accommodating changing conditions and system requirements.

One advantage of our implementation lies in the assignment of priorities to all scenarios within the Planner, Controller, and Recovery modules during creation. This design facilitates the addition of new scenarios to these modules without the need for modifications to the DM scenario. The incorporation of rules and strategies that can be dynamically applied to the robot's behavior during runtime makes our implementation highly adaptable and efficient. The feedback loop further allows for the selection of different strategies, resulting in entirely different behaviors without requiring code modifications during runtime [Chaychi, Zampunieris, and Reis, 2023].

Our implementation offers a robust solution, ensuring the system's effective response to changing circumstances without the need for code changes or system relaunches. The ability to dynamically adjust behavior using different rules and strategies at runtime enhances the system's adaptability and performance. This dynamic behavior allows for real-time changes during runtime, providing greater flexibility and efficiency to the overall system.

```
1
```

```
2     protected void actions Decision Making() {
3     super.actions();
4     saveLatestRcmdToDB(robotRcmdList);
5     saveLatestRcmdToDB(systemRcmdList);
6   }
7
8     private void saveLatestRcmdToDB(final CommandRcmdList
     rcmdList) {
9     boolean firstTreated = false;
10    final Iterator<CommandRecommendation> crListIterator =
     rcmdList.getList().iterator();
11    while (crListIterator.hasNext()) {
12      final CommandRecommendation rcmd = crListIterator.next();
13      Global_Vars.logger.finest(rcmd.toString());
14      if (!firstTreated) {
15        saveCommands2Robot(rcmd);
16        firstTreated = true;
17      }
18      rcmd.setTreated();
19    }
20  }
21
22  protected boolean rulesGeneration() {
23    createRule(this);
24    printCommands2Robot();
25    return true;
26  }
```

## 5.3 Database

In our MySQL database, we have several tables for efficiently storing data from both the ROS and Proactive Engine components. two tables are dedicated to storing data from LaserScan and Odometry received from the robot. These tables captures essential sensor information, such as distance measurements and robot pose, facilitating accurate navigation and mapping. On the other hand, we have separate tables for "command_recommendation" and "command2robot," which are utilized by the Decision Making process and saved to the database by the Proactive Engine. You can see our command tables in the Figure 5.3. These tables contain the final command recommendations and commands sent to the robot for execution, ensuring smooth and effective operation based on the system's decision-making process. To en-

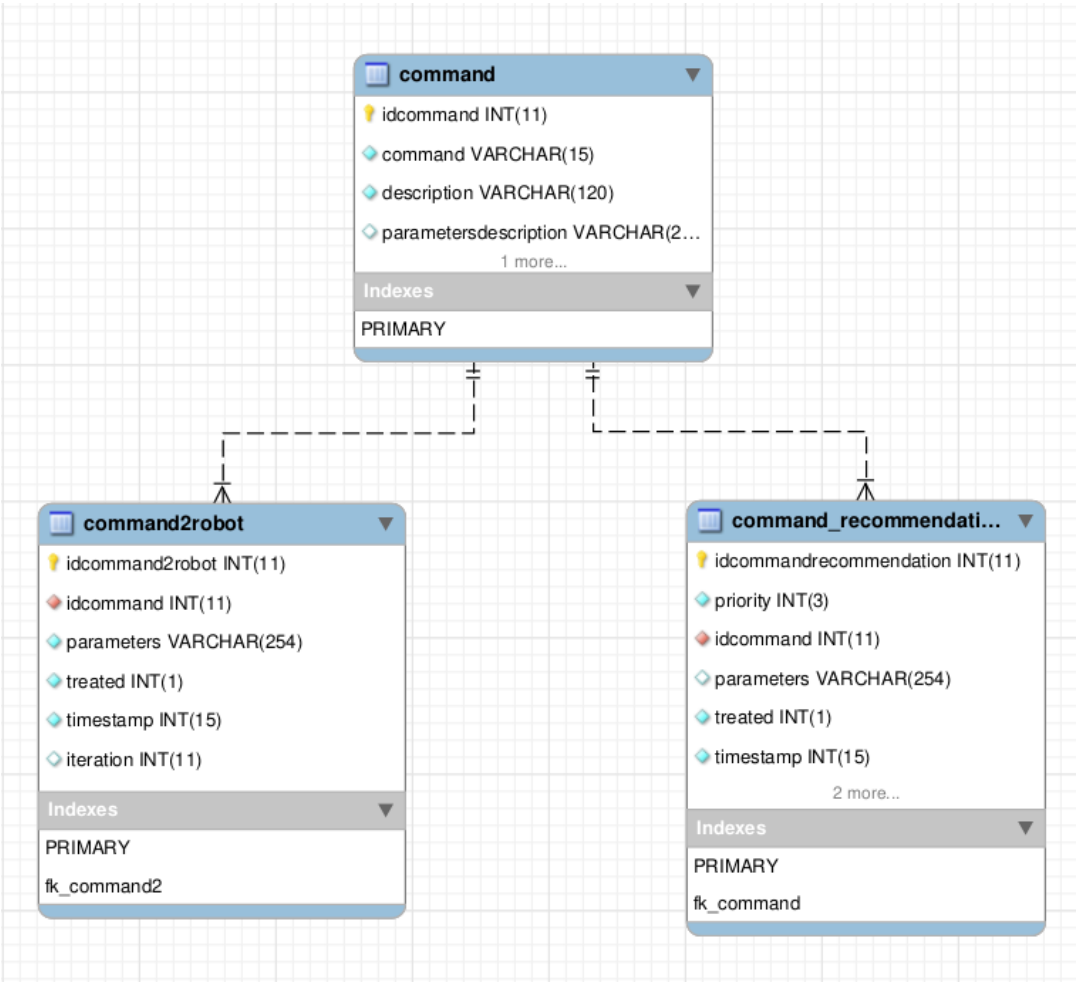


FIGURE 5.3: Command

hance communication between ROS and the Proactive Engine, we have implemented rules and conditions within the database. These rules serve as guidelines for data exchange and enable seamless coordination between the different components. By employing a structured database approach, we ensure efficient data management and communication, contributing to the overall effectiveness and reliability of our system.

### 5.3.1 Data from ROS

In our implementation, we subscribe to data from the robot, specifically Odometry and LaserScan data, and store it in our MySQL database regularly. This data is crucial for the system's operation and allows for efficient retrieval and analysis when making decisions and generating appropriate commands for the robot.

#### Odometry

Odometry is a crucial component in robotics, representing an estimate of the robot's position and velocity in free space. The pose information in the Odometry message is specified in the coordinate frame given by the header.fr ame_id. This frame provides context for interpreting the position and orientation of the robot. Additionally, the twist information in the Odometry message is specified in the coordinate frame given by the " child_frame_id. " The "child_frame_id" frame is used to define the robot's movement, capturing its linear and angular velocity.

In the Odometry message, the header contains essential metadata such as the timestamp ("stamp") and frame ID ("odom"). The child_frame_id section holds the position and orientation information, representing the robot's pose in the specified coordinate frame. The twist section contains data on the linear and angular velocity, reflecting the robot's movement in the specified coordinate frame. By combining these components, the Odometry message provides valuable insights into the robot's position, velocity, and movement within the given coordinate frames. This information is fundamental for navigation, mapping, and control tasks in robotic systems The example format of Odometry data is as follows:

```
header:
    stamp:
        sec: 3516
        nanosec: 858000000
    frame\_id: odom
```

```
child\_fram\_id: base\_footprint
pose:
    pose:
        position:
            x: 0.5220085108172701
            y: 0.5009689100135933
            z: 0.008716563288815664
        orientation:
            x: 0.000116762304784384
            y: 0.0007713307381056864
            z: -0.13695340825999733
            w: 0.990577182950136
    covarience:
    ...
twist:
    twist:
        linear:
            x: 5.831598723866704e-05
            y: 2.6196281658887204e-06
            z: 0.0
        angular:
            x: 0.0
            y: 0.0
            z: -0.0002702359079858924
    covariance:
    ...
```

**LaserScan**

In the LaserScan data, we have a header that includes essential information such as the timestamp for the acquisition time of the first ray in the scan and the frame ID, which is labeled as "laser sensor link." The angular measure is taken around the position Z-axis, with a counterclockwise direction (assuming Z is up) and the zero angle representing forward along the X-axis.

There are several parameters that define the characteristics of the LaserScan. "angle_min" indicates the starting angle of the scan in radians, while "angle_max" represents the ending angle of the scan in radians. The "angle_increment" parameter specifies the angular distance between consecutive measurements in radians. The "time_increment" parameter denotes the time between individual measurements in seconds and is used for interpolating the position of 3D points when the scanner is in motion. Additionally, "scan_time" indicates the time between consecutive scans in seconds.

For the actual laser measurements, we have "range_min" and "range_max" parameters, which define the minimum and maximum range values that the scanner can detect. The "ranges" data contains the measured distances from the scanner to the detected objects. Lastly, the "intensities" parameter holds the intensity data, which might be empty if the scanner does not provide intensity information. The example format of LaserScan data is as follows:

```
header:
    seq:5
    stamp:
        secs: 2829
        nsecs: 69000000
    frame_id: "laser\_sensor\_link"
angle_min: -1.57079994678
angle_max: 1.57079994678
angle_increment: 0.00436940183863
time_increment: 0.0
scan_time: 0.0
range_min: 0.10000000149
range_max: 30.0
range: [inf,...,2.46343994140625,...,inf]
```

### 5.3.2 Data from Proactive Engine

In our MySQL database, we have multiple tables dedicated to the Proactive Engine's functionalities. Each scenario within the Proactive Engine has its own table for storing the chosen strategies and recommendations. These tables capture the recommended actions generated by each scenario, enabling a comprehensive view of the system's decision-making process.

Furthermore, we have a separate table named "command2robot," which serves as the final destination for the decision-making process. Once the Decision Making scenario makes its final decision, it sends the corresponding command to the "command2robot" table within the database. From there, the command is retrieved by the database and transmitted to the robot for execution.

## 5.4 Robot Operating System

In our ROS implementation, we utilize the latest version, ROS Foxy, to take advantage of its updated features and capabilities.

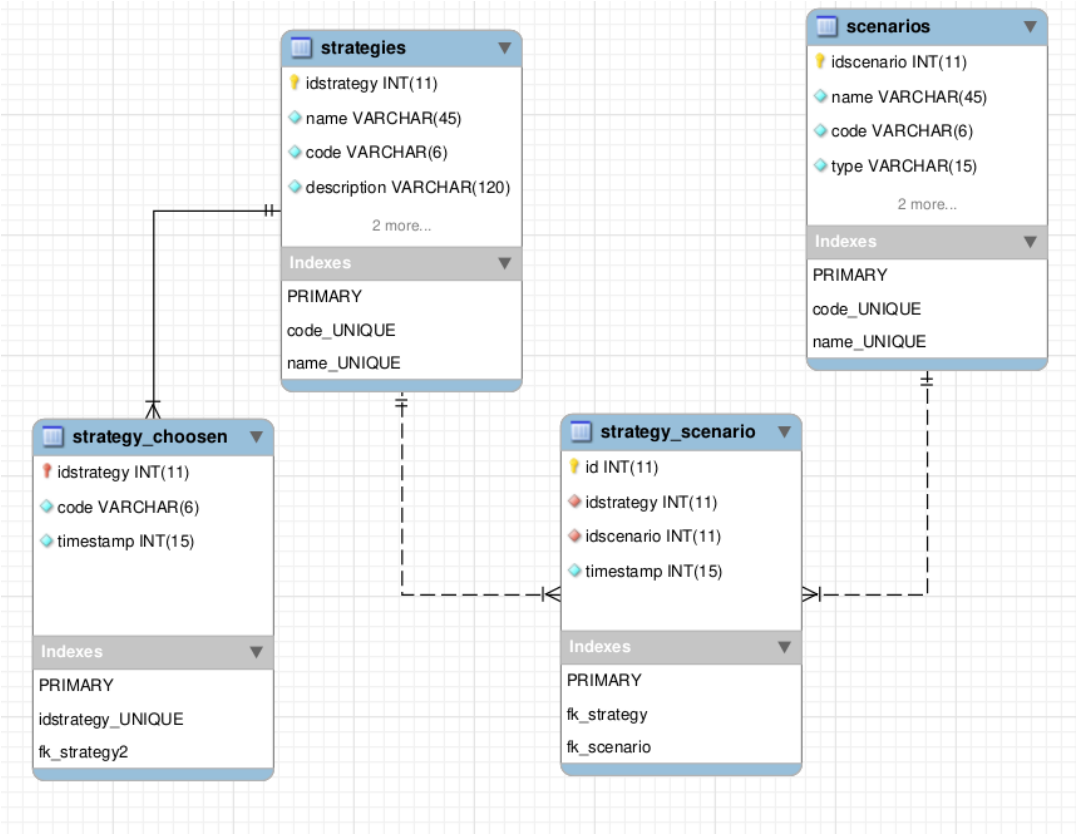


FIGURE 5.4: Local Storage

Our system consists of both subscriber and publisher nodes for effective communication with the robot and the database.

To ensure separation of concerns, we have two subscriber nodes: one for receiving LaserScan data and another for Odometry data. The LaserScan subscriber processes the data and extracts the essential range information, which is then saved into the corresponding table in the database. On the other hand, the Odometry subscriber receives data and stores the position, orientation, twist linear, and angular information into the appropriate table in the database.

In the publisher node, we read the commands from the database and convert them into the required twist format, represented as [ Linear.X, Linear.Y, Linear.Z; Angular.X, Angular.Y, Angular.Z]. The transformed commands are then published to the robot, enabling it to execute the desired actions based on the received commands.

By using ROS Foxy and structuring the communication between the subscriber, publisher, and the database, we ensure efficient data exchange and seamless interaction with the robot. This implementation allows for smooth operation and effective control of the robot's movements and behaviors.

# Chapter 6

# Comparison

## 6.1 Introduction

In order to evaluate the outcomes of our implementation, we will perform a comparative analysis between our solution and Navigation 2. This analysis will encompass several aspects, such as compile time, runtime, code modification and extension, and dynamic system changes.

Software metrics play a pivotal role in facilitating comparison, enabling a comprehensive assessment of diverse aspects of software. In the scope of my thesis, which emphasizes code extension, reusability, and maintenance, the comparison with Navigation 2 involves scrutinizing code complexity, quality, and performance metrics. The researched metrics encompass size[Nguyen et al., 2007], complexity[Henderson-Sellers, 1995], coupling, cohesion[Cai et al., 2014], maintainability[Abdullah, 2017], reliability, performance, and security[Lee, 2014]. Additionally, dynamic coupling metrics[Arisholm, Briand, and Foyen, 2004], dynamic cohesion, and dynamic complexity are considered in our research [Kumar Chhabra and Gupta, 2010].

In our meticulous examination of software metrics, we explore their practical applications in both C++ and Java, aligning with our chosen programming languages for the robot and Proactive Engine parts. For C++, tools like Datadog[1], and CodeMR[2] stand out as exemplary choices for measuring various code quality aspects, providing detailed analyses of complexity, coupling, cohesion, and maintainability within the C++ codebase. On the Java side SonarQube[3], Eclipse[4], Datadog and CodeMR remain robust tools for comprehensive code analysis, showcasing their adaptability across different programming languages and contributing to a thorough examination

[1]https://www.datadoghq.com/
[2]https://www.codemr.co.uk/
[3]https://www.sonarqube.org/
[4]https://www.eclipse.org/

of Java code, ensuring consistent metric measurements across the entire software system.

For our chosen software metrics, CodeMR is utilized for compile-time analysis, offering valuable insights into the code structure during compilation. Its versatility in handling both C++ and Java makes it suitable for our hybrid system. Datadog takes the spotlight for run-time metrics, providing real-time monitoring and performance analysis. Its compatibility with both C++ and Java enables a holistic evaluation of the system's performance and behavior during execution.

Through the utilization of these tools, we aim to conduct a robust comparison between our system and Navigation 2, ensuring effective measurement and analysis of chosen metrics in both programming languages. The combined use of CodeMR and Datadog, along with other supporting tools, forms a comprehensive approach to evaluating our system's codebase, reusability, and maintenance against the industry-standard Navigation 2. Within these software metrics, specific additional metrics are considered, allowing for a detailed measurement and comparison of our system with Navigation 2. Subsequent sections provide a detailed explanation of each software tool and its associated software metrics.

For comparing the compile time, we will utilize the CodeMR[5] tool, while the runtime analysis will be utilized using Datadog[6]. These tools will offer valuable insights into the performance and behavior of both software systems. By utilizing CodeMR and Datadog, our objective is to gain a comprehensive understanding of the similarities and differences between the two systems in terms of their compilation and execution processes.

## 6.2 Compile Time

We have used the CodeMR software tools to compare our system with Navigation 2 during the compilation phase. All definitions and synonyms used in this section are sourced from the official CodeMR webpage. CodeMR is a robust software quality and static code analysis tool that helps software companies develop high-quality products with improved code. It allows us to visualize code metrics and high-level quality attributes such as Coupling, Lack of Cohesion, Complexity, and Size for both C++ and Java. Since our

[5]https://www.codemr.co.uk/
[6]https://www.datadoghq.com/

Proactive Engine is implemented in Java and Navigation 2 is implemented in C++, and CodeMR supports both programming languages, we have chosen to utilize this tool for analysis and comparison purposes.

### 6.2.1 Software Quality Attributes

External quality of software refers to the noticeable problems that arise, while the true causes lie within the internal quality attributes. These internal attributes include various factors, but coupling, complexity, cohesion, and size are the key elements that strongly influence the overall quality of a software system(Quoted from the CodeMR webpage).

#### Coupling

In software engineering, coupling refers to the degree of interdependence between software modules. It is a measure of how closely connected two routines or modules are, and represents the strength of the relationships between them[7].

Coupling between two classes, A and B, can occur in several ways, including when A has an attribute that refers to B, A calls on services of an object B, A has a method that references B (via return type or parameter), A has a local variable of type B, or A is a subclass of (or implements) class B. Tightly coupled systems typically exhibit the following characteristics: changes made to one class usually force a ripple effect of changes in other classes, which can require more time and effort due to the increased dependency. Moreover, tightly coupled systems may be harder to reuse because dependent classes must also be included.

CodeMR software tools use the Coupling Between Object Classes (CBO) as the basis for measuring coupling. CBO is calculated by counting the number of other classes that use the attributes or methods of a given class, as well as the number of classes whose attributes or methods are used by the given class. Inheritance relations are excluded from this calculation (Quoted from the CodeMR webpage). High coupling can make the code more difficult to maintain because changes made to other classes can also cause changes in

[7]https://en.wikipedia.org/wiki/Coupling_(computer_programming)

the given class. Moreover, highly coupled classes are less reusable and require more testing effort. The levels of Coupling Between Objects (CBO) can be categorized into five different groups:

- Low: CBO <= 5
- Low medium: (CBO < 5) AND (CBO <= 10)
- Medium high: (CBO > 10) AND (CBO <= 20)
- High: (CBO > 20) AND (CBO <= 30)
- Very high: CBO > 30

**Lack of Cohesion**

Cohesion refers to the measure of how well the methods of a class are related to each other. High cohesion (low lack of cohesion) is generally preferred because it is associated with several desirable traits of software, including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as difficulty in maintaining, testing, reusing, or understanding the code(Quoted from the CodeMR webpage).

LCOM (Lack of Cohesion of Methods) is a metric used to evaluate the cohesion of a class. Low cohesion means that the class implements more than one responsibility, and a change request for one responsibility will result in changes to the entire class. Lack of cohesion also affects understandability and implies that classes should be split into two or more subclasses. LCOM3 is a variation of LCOM and is defined as follows:

$$LCOM3 = (m - sum(mA)/a)/(m - 1)$$

Where m is the number of methods in the class, a is the number of variables in the class (both shared and non-shared), and mA is the number of methods that access a variable. LCOM3 varies between 0 and 2, and values between 1 and 2 are considered alarming. In a normal class, LCOM3 varies between 0 (high cohesion) and 1 (no cohesion). LCOM3=0 indicates the highest possible cohesion, where each method accesses all variables. LCOM3=1 indicates extreme lack of cohesion, and in this case, the class should be split.

If there are variables that are not accessed by any of the class's methods, LCOM3 will be greater than 1. This indicates a design flaw, and the class

is a candidate for rewriting as a module. Alternatively, the class variables should be encapsulated with accessor methods or properties, and dead variables should be removed. If there is only one method in a class, LCOM3 is undefined. If there are no variables in a class, LCOM3 is also undefined, and it is displayed as zero[8].

The levels of cohesion can be classified into five categories based on their numerical values. These categories are as follows:

- Low: LCAM <= 0.6
- Low medium: (LCAM > 0.6) AND (LCAM <= 0.7)
- Medium high: (LCAM > 0.7) AND (LCAM <= 0.8)
- High: (LCAM > 0.8) AND (LCAM <= 0.9)
- Very high: LCAM > 0.9

**Complexity**

Complexity can be measured using several metrics, including Weighted Meth od Count (WMC), Response For a Class (RFC), and Depth of Inheritance Tree (DIT). WMC is the weighted sum of all methods in a class, where the complexity of each method is usually taken as 1. A high WMC indicates a more complex class, which can increase development, maintenance, and testing effort. Inheritance can also affect WMC, as all methods in the base class are represented in its child classes. Highly domain-specific classes with a high number of methods are less reusable and tend to be more prone to defects and changes.

RFC measures the number of methods that can potentially be invoked in response to a public message received by an object of a particular class. If the RFC value is high, the class is considered more complex and may be highly coupled to other classes, requiring more testing and maintenance effort.

DIT indicates the position of a class in the inheritance tree, with a 0 value for root and non-inherited classes. For multiple inheritance, the metric shows the maximum length. Deeper classes in the inheritance tree may be more complex to develop, test, and maintain, as their behavior is harder to predict (Quoted from the CodeMR webpage).

To interpret these metrics, we can use the following scale:

- Low: (WMC <=20) OR (RFC <= 50) OR (DIT <=1)

[8]https://www.aivosto.com/project/help/pmoo-cohesion.html.LCOM4

- Low medium: (20 < WMC <= 50) OR (50 < RFC <= 100) OR (1 < DIT <= 3)
- Medium high: (50 < WMC <= 101) OR (100 < RFC <= 150) OR (3 < DIT <= 10)
- High: (101 < WMC <= 120) OR (150 < RFC <= 200) OR (10 < DIT <= 20)
- Very high: (WMC > 120) OR (RFC > 200) OR (DIT > 20)

**Size**

Size is an important aspect of software quality, and two common metrics used to measure it are Line of Code (LOC) and Number of Methods (NOM).

LOC measures the number of nonempty, non-commented lines of code in the body of a class. A high LOC can indicate a more complex class and increase the potential for errors.

NOM measures the number of methods in a class. A high NOM can also indicate a more complex class, which can lead to more difficult maintenance and testing (Quoted from the CodeMR webpage).

To interpret these metrics, we can use the following scale:

- Low: (LOC < 50) OR (NOM <= 20)
- Low medium: (50 < LOC 300 ) OR ( 20 < NOM <= 30)
- Medium high: (300 < LOC <= 900) OR (30 < NOM <= 40)
- High: (900 < LOC <= 1500) OR (40 < NOM <= 50)
- Very high: (LOC > 1500) OR (NOM > 50)

### 6.2.2 Exploring Attribute Visualizations

CodeMR provides various visualizations, including Overview, Metric Distribution, Package Structure, Sunburst, Package Dependency, TreeMap, and Project Outline. Our plan is to utilize these visualizations to compare two systems by displaying their respective visual representations (Quoted from the CodeMR webpage). As depicted in Figure (6.1), the color of the chart shapes corresponds to the metric of the respective software entity. Metrics are categorized into five levels: low, low-medium, medium-high, high, and very-high. The use of red indicates a high value for the selected metric, while the use of green represents the lowest values.

FIGURE 6.1: Low -> High

### Overview

The Overview tab in the codeMR model editor provides general information about the extracted project, including the total number of lines of code, number of classes, number of packages, number of problematic classes, and number of highly problematic classes. Furthermore, the tab presents a pie chart that shows the percentage of metric levels for the selected metric, proportional to the code size of the classes in each level. Our results show that we have selected C3, which represents the coupling, cohesion, and complexity quality attributes and has the maximum value across the coupling, cohesion, and complexity matrices (Quoted from the CodeMR webpage). In the figure(6.2) you see the result of proactive engine and in the figure (6.3) you see the result of Navigation 2.

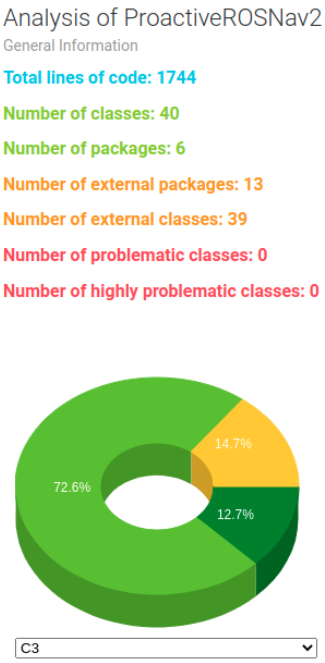


FIGURE 6.2: Overview Analysis of Proactive Engine

Based on the provided figures for the Proactive Engine and Navigation 2, it is noticeable that the Proactive Engine project has significantly fewer lines of code compared to the Navigation 2 project. Specifically, the Proactive Engine project consists of 1722 lines of code, while navigation2 has 9387 lines

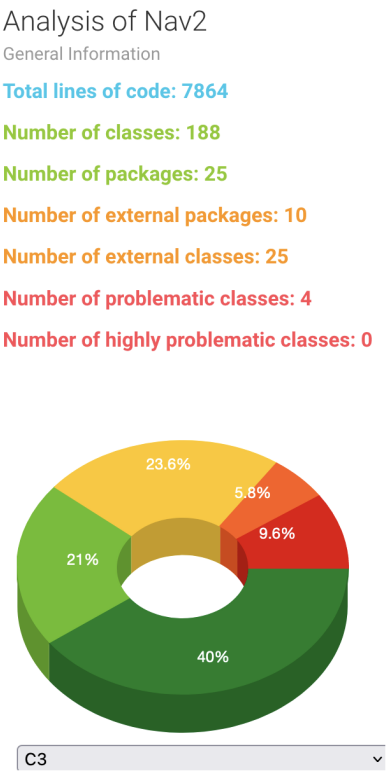


FIGURE 6.3: Overview Analysis of Navigation 2

of code. This makes the Proactive Engine project approximately five times smaller than Navigation 2. In terms of the C3 metric, which measures coupling, complexity, and cohesion, the Proactive Engine project demonstrates a relatively low level of complexity. Only 14.7% of the project falls under the medium-high C3 category. On the other hand, Navigation 2 exhibits a higher level of C3, with 21.4% categorized as medium-high, 9.3% as high, and 8% as very high C3.

### Metric Distribution

The Metric Distribution Tab displays the percentage of metric levels for each metric in pie charts. The size of the pie chart slices is proportional to the code size of the corresponding classes for each level. This visualization provides a detailed chart for Complexity, Coupling, Lack of Cohesion, and Size. In the figure(6.4) you see the result of Proactive Engine and in the figure (6.5) you see the result of Navigation 2.

Based on the provided figure comparing the Proactive Engine and Navigation 2 projects, it is noticeable that the proactive engine project performs

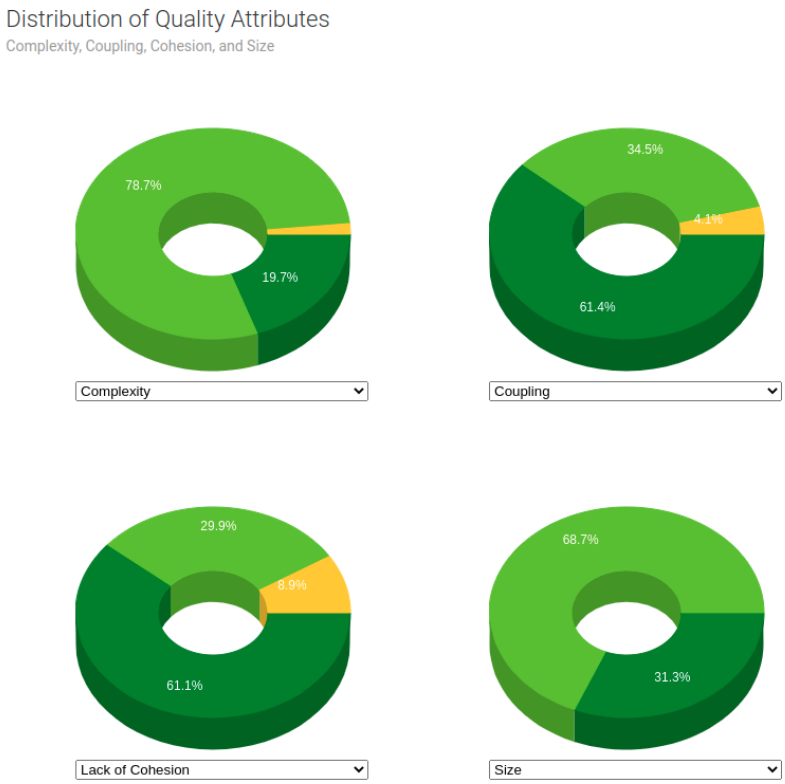


FIGURE 6.4: Metric Distribution of Proactive Engine

significantly better than the Navigation 2 project. Specifically, in terms of complexity, the Proactive Engine project has a low percentage of medium-level complexity, with the majority of the project having low and low-medium complexity. On the other hand, Navigation 2 has 8% very high complexity and 13.5% medium-level complexity. In terms of coupling, the Proactive Engine project has only 4.1% medium-high coupling, with the rest having low and low-medium coupling. In contrast, Navigation 2 has 6.1% medium-high coupling. Regarding lack of cohesion, the Proactive Engine project has 8.9% medium-high level of lack of cohesion, while Navigation 2 has 9.3% high and 18.1% medium-high level of lack of cohesion. Finally, concerning project size, the proactive engine project has a low and medium-low size, while Navigation 2 has a high size of 3.6% and a medium-high size of 18.7%.

### Package Structure

The Package Structure feature is a visual representation of a project's packages and encapsulated classes displayed in a hierarchical manner using a circle pack layout. This feature has several properties, including circle sizes that are proportional to the size of the represented software entity, and circle


FIGURE 6.5: Metric Distribution of Navigation 2

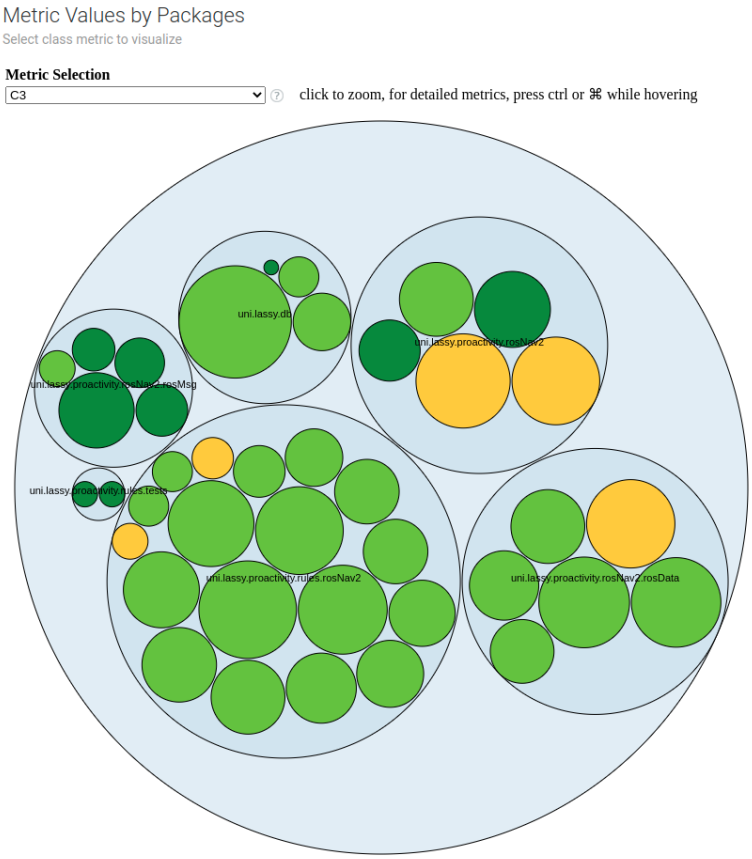


FIGURE 6.6: Package Structure of Proactive Engine

colors that represent the level of the selected metric. Additionally, when the Metric Chart option is selected, hovering over a class displays its metrics in the CodeMR Metric Chart. The Package Structure feature is also zoomable, allowing users to change the zoom level by clicking the circles. Overall, this feature provides a comprehensive and visually appealing way to understand a project's package structure and metrics (Quoted from the CodeMR webpage).

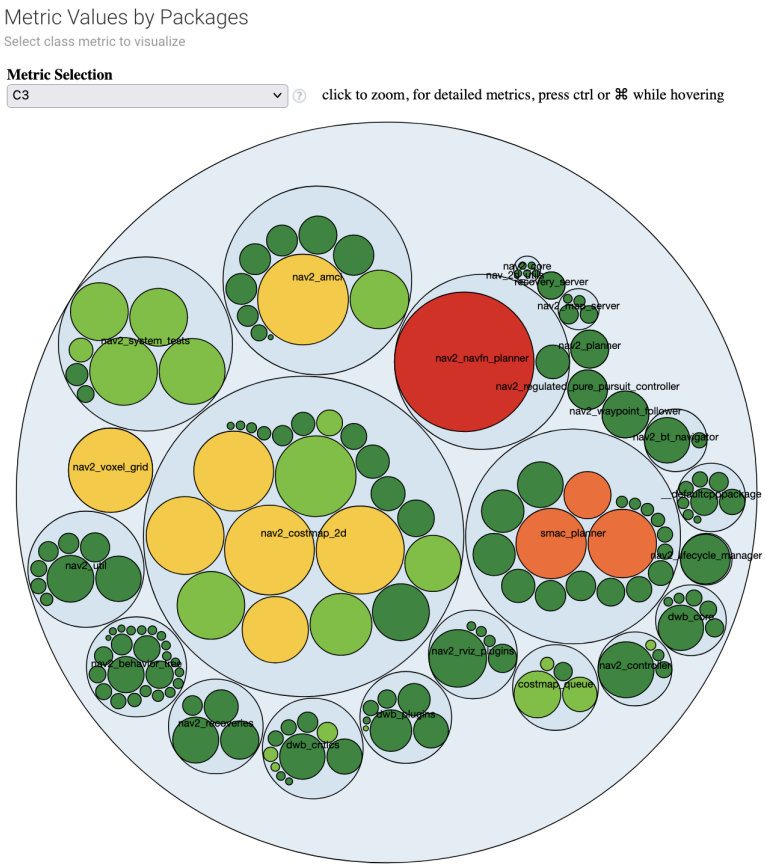


FIGURE 6.7: Package Structure of Navigation2

Based on the provided figures for the Proactive Engine and Navigation 2, Figure (6.6) illustrates the package structure and encapsulated packages of the Proactive Engine, taking into account the C3 metric, which measures coupling, complexity, and cohesion. Figure (6.7) depicts the package structure and encapsulation in Navigation 2. In terms of size, the "nav2_navfn_planner" package stands out as the largest package in both Navigation 2 and the Proactive Engine. However, the other classes in Navigation 2 are either similar in size or smaller when compared to the Proactive Engine. When considering complexity levels indicated by colors, Navigation 2 exhibits a very high complexity in the "nav2_navfn_planner" package. Additionally, there are three

other package with high complexity. In contrast, the Proactive Engine does not have any packages with very high or high complexity.

### Sunburst

The Sunburst view is a useful tool for displaying hierarchical data. It uses a radial layout, with the root node of the tree at the center and leaves on the circumference. The angle of each arc corresponds to the size (in lines of code) of the elements it represents. Users can select class, package, and project metrics separately. The Sunburst view has several properties, including angles that are proportional to the size of the represented software entity and colors that represent the level of the selected metric. When the Metric Chart option is selected, hovering over a class displays its metrics in the CodeMR Metric Chart. The Sunburst view is also zoomable, allowing users to change the zoom level by clicking the arcs (Quoted from the CodeMR webpage). In the figure(6.8) you see the result of Proactive Engine and in the figure (6.9) you see the result of Navigation 2.

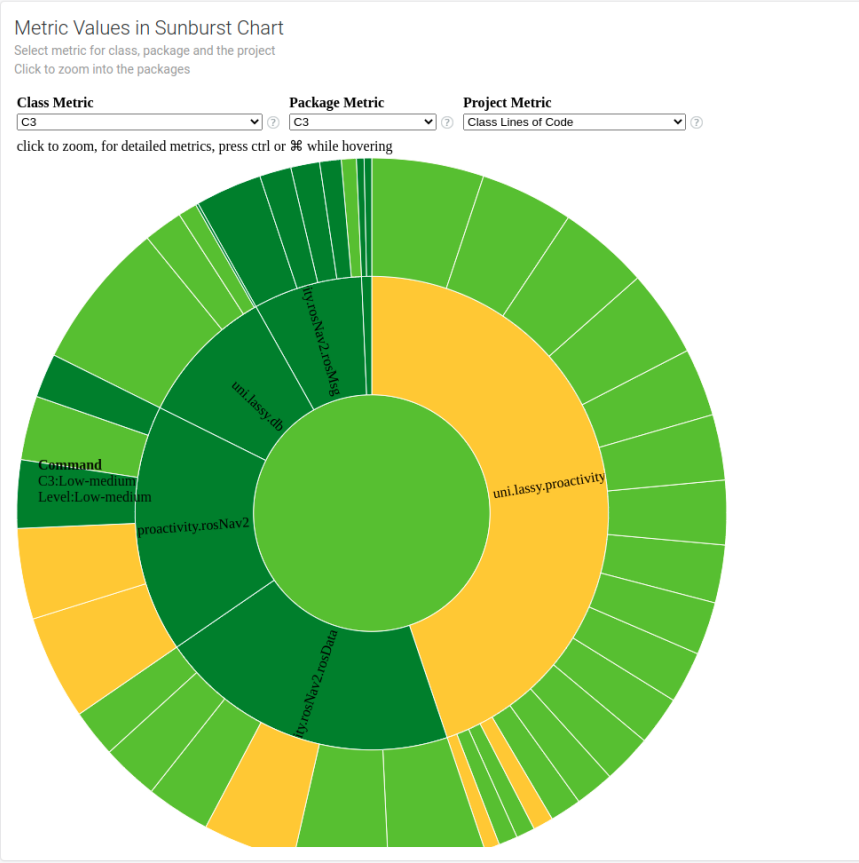


FIGURE 6.8: Sunburst Chart of Proactive Engine

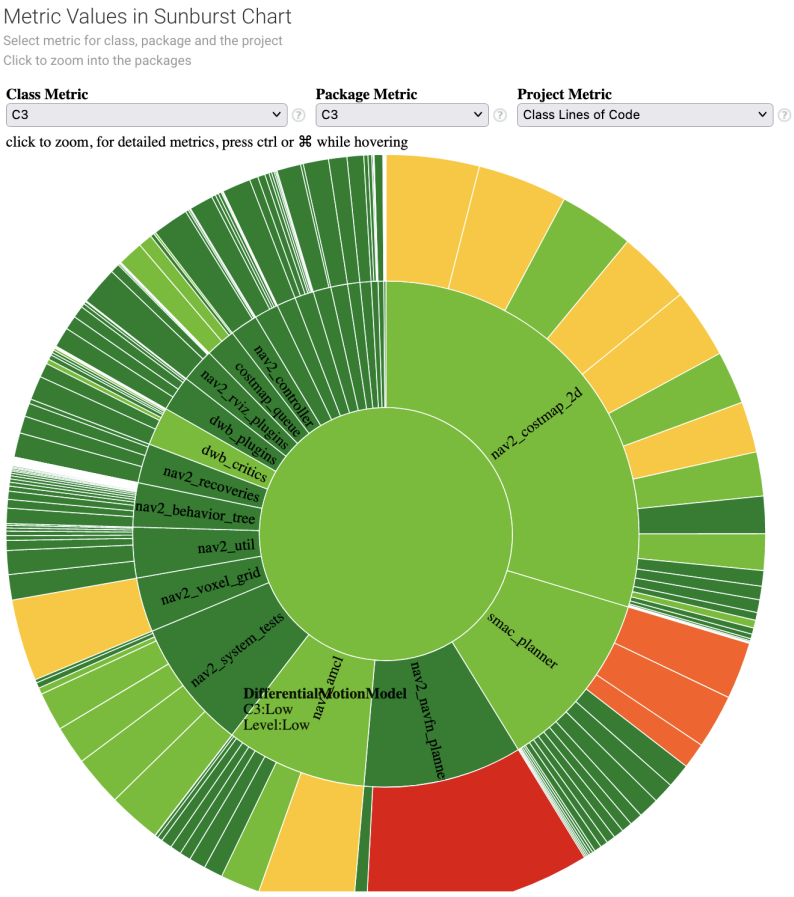


FIGURE 6.9: Sunburst Chart of Navigation 2

Based on the provided figures comparing the Proactive Engine, and Navigation 2, the Sunburst charts offer more detailed information about each class, including their hierarchical relationships. In the Proactive Engine Sunburst chart, the "proactivity" package stands out with a medium-high C3 level. Clicking on each class provides a more detailed analysis within the chart. In the Navigation 2 project, the "nav2_navfn_planner" package exhibits a very high C3 level, and there are several other packages with high and medium-high C3 levels as well.

### Package Dependency

The Package Dependency view arranges packages radially, connecting them with thick curves. The thickness of the curve represents the frequency of relations between two packages. If a chord is tapered, it indicates that there are more relations from a given package compared to the relations it receives. In the CodeMr software, by hovering over the chord between packages, you

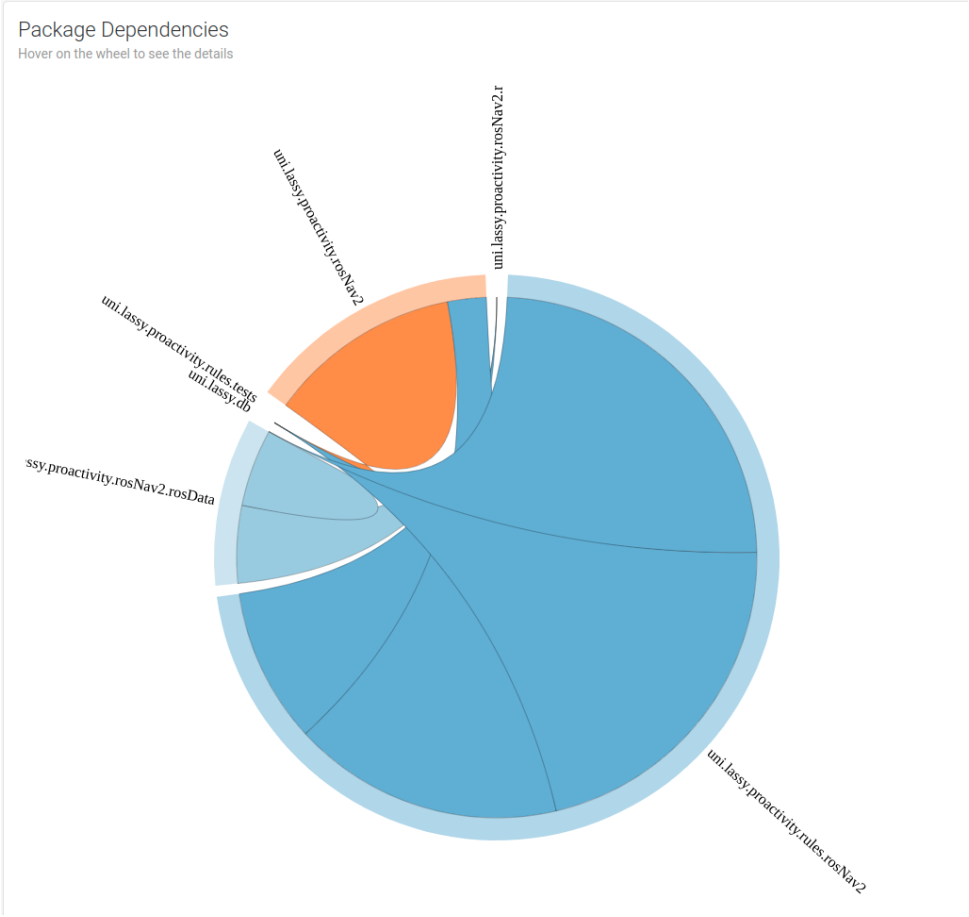


FIGURE 6.10: Package Dependency of Proactive Engine

can view the number of relations in the tooltip (Quoted from the CodeMR webpage).

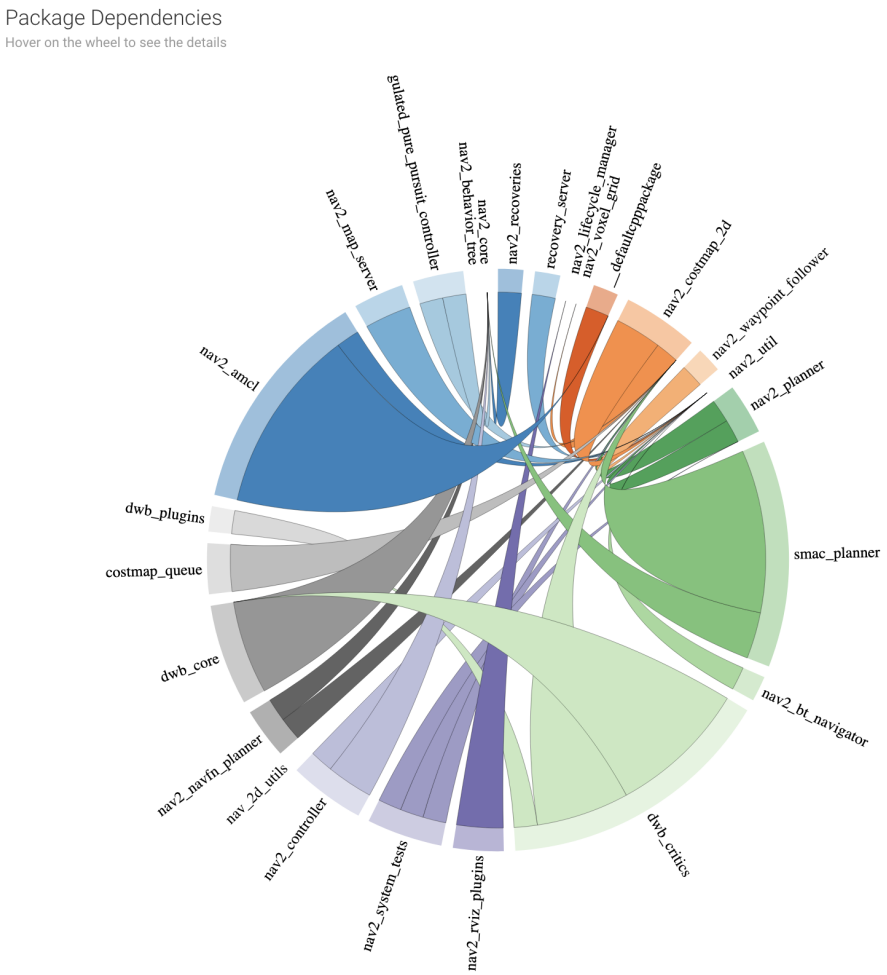


FIGURE 6.11: Package Dependency of Navigation2

Based on the provided figures comparing the dependency of Proactive Engine and Navigation 2, it is evident that the packages in Navigation 2 have a higher level of dependency. Each package in Navigation 2 relies on multiple other packages. On the other hand, it is worth noting that Proactive Engine has fewer packages, and each package has fewer dependencies.

### TreeMap

A TreeMap is a visualization technique used to represent software entities based on their size, typically measured in lines of code. It divides an area into rectangles, with each rectangle representing a software entity. The size of the rectangle corresponds to the size of the entity it represents. A key feature of TreeMap is that the area of each rectangle is proportional to the

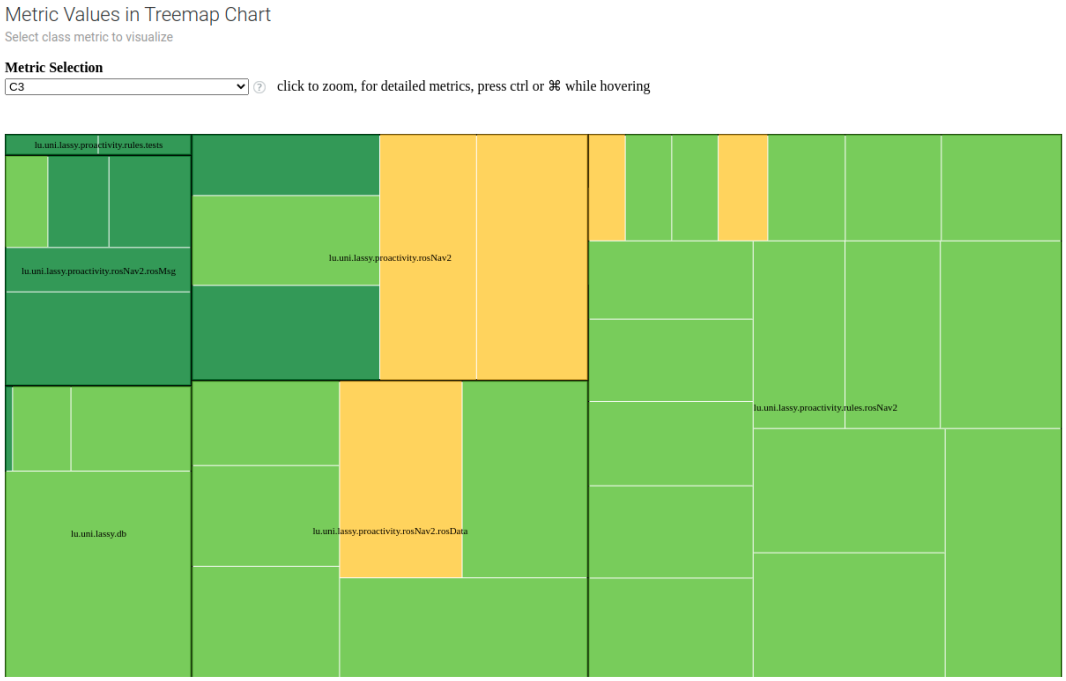


FIGURE 6.12: TreeMap of Proactive Engine

entity's size. The color of the rectangles indicates the level of a selected metric, making it easy to identify areas that need attention. If the Metric Chart option is chosen, hovering over a class provides access to its metrics in the CodeMR Metric Chart. One advantage of TreeMap is its zoom capability, allowing users to change the zoom level by clicking on the rectangles. This enables detailed examination or a broader overview of the entire software system. Overall, TreeMap is an effective visualization tool for analyzing complex software systems and identifying areas that require further investigation (Quoted from the CodeMR webpage). In the figure(6.12) you see the result of Proactive Engine and in the figure (6.13) you see the result of Navigation 2.

Based on the provided figures comparing the Proactive Engine and Navigation 2 using the TreeMap visualization, the size of the rectangles represents lines of code. Considering the figure and our previous measurements of size, it is noticeable that the Proactive Engine outperformed ROS. In terms of the color of the rectangles, it is worth noting that Navigation 2 has a considerable amount of C3, with several instances of high and medium-high levels. However, in the Proactive Engine, we only have a few medium-level instances of C3.

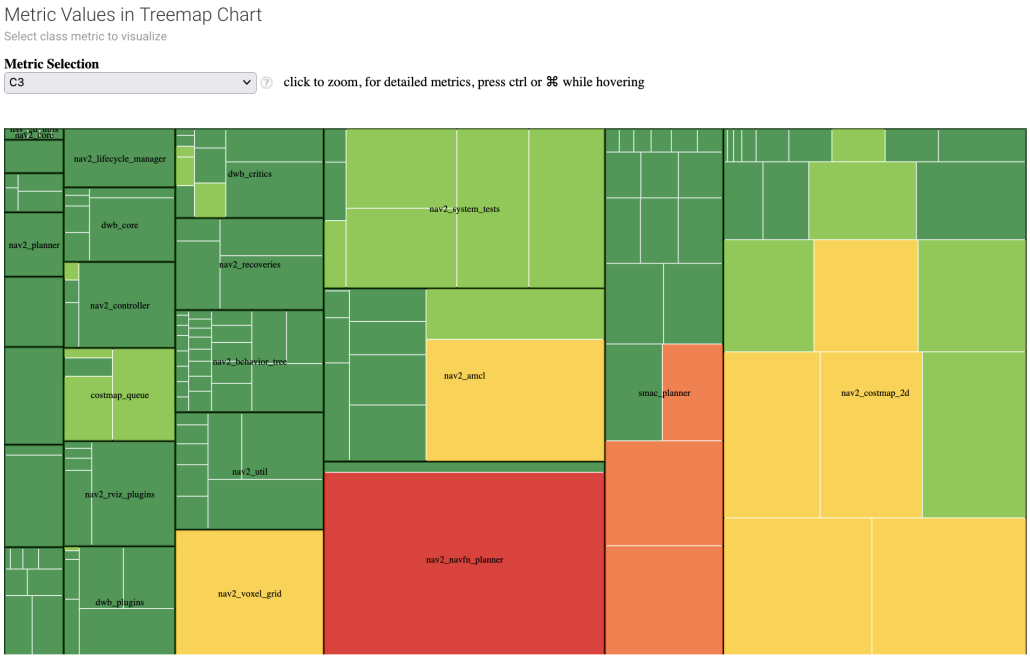


FIGURE 6.13: TreeMap of Navigation2

### Project Outline

The Outline view presents the quality attributes and metrics of software elements in a structured tree table. This view allows you to sort the elements by different metrics and quality attributes, enabling you to quickly identify problematic elements. Additionally, you can easily jump to the source code of any selected element. This feature helps in the speedy resolution of issues and improves the overall quality of the software system (Quoted from the CodeMR webpage).

In the figure(6.14) you see the result of Proactive Engineand in the figure (6.15) you see the result of Navigation 2.

| Element | Quality Attributes | LOC | Coupling | Complexity | Size | Lack of Cohesic | DIT | WMC | LOC |
|---|---|---|---|---|---|---|---|---|---|
| ▲ ProactiveROSNav2 | | | | | | | | | |
| lu.uni.lassy.db | | 166 | low | low | low | low | | 52 | 166 |
| lu.uni.lassy.proactivity.rosNav2 | | 295 | low | low | low-mediun | low | | 93 | 295 |
| lu.uni.lassy.proactivity.rosNav2.r | | 359 | low | low | low-mediun | low | | 107 | 359 |
| lu.uni.lassy.proactivity.rosNav2.r | | 130 | low | low | low | low | | 45 | 130 |
| lu.uni.lassy.proactivity.rules.rost | | 782 | medium-hig | low-mediun | medium-hig | low | | 202 | 782 |
| lu.uni.lassy.proactivity.rules.test | | 12 | low | low | low | low | | 4 | 12 |

FIGURE 6.14: Project Outline of Proactive Engine

Based on the provided figures and comparing the Proactive Engine and Navigation2 using the project outline view, the results are consistent with the

| Element | Quality Attributes | LOC | Co ing | Complexity | Size | Lack of Cohesion | NOC | WMC | LOC |
|---|---|---|---|---|---|---|---|---|---|
| Nav2 | | | | | | | | | |
| __defaultcpppackage | | 95 | low | low | low-medium | low | | 6 | 95 |
| costmap_queue | | 151 | low | low | low | low | | 37 | 151 |
| dwb_core | | 121 | low | low | low-medium | low | | 23 | 121 |
| dwb_critics | | 194 | low-medium | low | medium-high | low | | 46 | 194 |
| dwb_plugins | | 179 | low | low | low-medium | low | | 44 | 179 |
| nav_2d_utils | | 0 | low | low | low | low | | 0 | 0 |
| nav2_amcl | | 709 | low-medium | low | medium-high | low | | 95 | 709 |
| nav2_behavior_tree | | 221 | low | low | medium-high | low | | 56 | 221 |
| nav2_bt_navigator | | 89 | low | low | low | low | | 11 | 89 |
| nav2_controller | | 140 | low | low | low | low | | 24 | 140 |
| nav2_core | | 10 | low | low | low-medium | low | | 5 | 10 |
| nav2_costmap_2d | | 2332 | low | low-medium | medium-high | low | | 450 | 2332 |
| nav2_lifecycle_manager | | 96 | low | low | low | low | | 17 | 96 |
| nav2_map_server | | 34 | low | low | low | low | | 5 | 34 |
| nav2_navfn_planner | | 798 | low | low | low-medium | low | | 168 | 798 |
| nav2_planner | | 56 | low | low | low | low | | 10 | 56 |
| nav2_recoveries | | 199 | low | low | low | low | | 34 | 199 |
| nav2_regulated_pure_pursuit_controller | | 61 | low | low | low | low | | 8 | 61 |
| nav2_rviz_plugins | | 171 | low | low | low | low | | 32 | 171 |
| nav2_system_tests | | 655 | low | low | low-medium | low | | 104 | 655 |
| nav2_util | | 254 | low | low | medium-high | low | | 50 | 254 |
| nav2_voxel_grid | | 274 | low | low | low-medium | low | | 71 | 274 |
| nav2_waypoint_follower | | 85 | low | low | low | low | | 14 | 85 |
| recovery_server | | 29 | low | low | low | low | | 5 | 29 |
| smac_planner | | 911 | low-medium | low | medium-high | low | | 188 | 911 |

FIGURE 6.15: Project Outline of Navigation 2

previous view.

### 6.2.3 Summary

In conclusion, the comparison between the Proactive Engine and Navigation 2 revealed significant differences in terms of code size, complexity, coupling, and lack of cohesion. The Proactive Engine project had a significantly smaller codebase and demonstrated lower complexity levels, with a lower percentage of medium-high complexity, coupling, and lack of cohesion compared to Navigation 2. The project size of the Proactive Engine was also relatively smaller. The figures and visualizations further supported these findings, showcasing the differences in package structures, class sizes, and C3 levels between the two projects. Overall, the Proactive Engine outperformed Navigation 2 in terms of codebase efficiency and complexity management.

## 6.3 Runtime

We utilized Datadog software tools to analyze our system with Navigation 2 in runtime. The runtime measurements were conducted on a PC Dell system featuring an Intel Core i7 6700 processor with 8 threads and 16 GB of

RAM. The experiments were executed under the Ubuntu 20.04 LTS operating system for consistent runtime assessments. Datadog is an essential platform that focuses on monitoring and enhancing the security of cloud applications. By integrating end-to-end traces, metrics, and logs, it offers comprehensive observability for your applications, infrastructure, and third-party services. These powerful capabilities enable businesses to effectively secure their systems, minimize downtime, and optimize the overall user experience for their customers (Quoted from the Datadog webpage). To launch Datadog, we need to run the Datadog Agent[9], which is a software that runs on our host. It is an open-source tool available on GitHub. The Datadog Agent collects events and metrics from the host and sends them to Datadog, where we can analyze and monitor the data. Datadog can be accessed through a web browser for data monitoring. Using Datadog, we conducted measurements for CPU usage, CPU usage by process, and memory usage. To ensure accuracy, we performed 10 measurements for each metric. However, for presentation purposes, we are showcasing a typical single execution. Below, you will find the results and a comparison between the proactive engine and navigation 2 based on these measurements.

### 6.3.1 CPU Usage

CPU time refers to the duration during which a central processing unit (CPU) is actively engaged in processing instructions for a computer program or operating system. It specifically focuses on the time dedicated to executing program instructions, excluding factors such as waiting for input/output (I/O) operations or entering low-power (idle) mode. CPU time is typically measured in clock ticks or seconds. To gauge the efficiency of CPU utilization, it is often valuable to express CPU time as a percentage of the CPU's total capacity. This measurement is referred to as CPU usage and indicates the proportion of time the CPU is actively performing tasks relative to its maximum capability. By monitoring CPU usage, one can gain insights into the workload and effectiveness of the CPU in executing instructions[Ehrhardt, 2010]

To measure the CPU usage of the Proactive Engine, the following steps were taken:

1. The Datadog agent was launched at 10:45 minutes.

[9]https://docs.datadoghq.com/agent/

2. A waiting period of 5 minutes was observed to ensure system stability.

3. At 10:50, Gazebo (simulation software) and Eclipse (IDE) to run the Proactive Engine were launched.

4. Also, as part of our implementation, we utilized ROS so at this point ROS nodes were launched to facilitate the exchange of data between ROS and the Proactive Engine through a database.

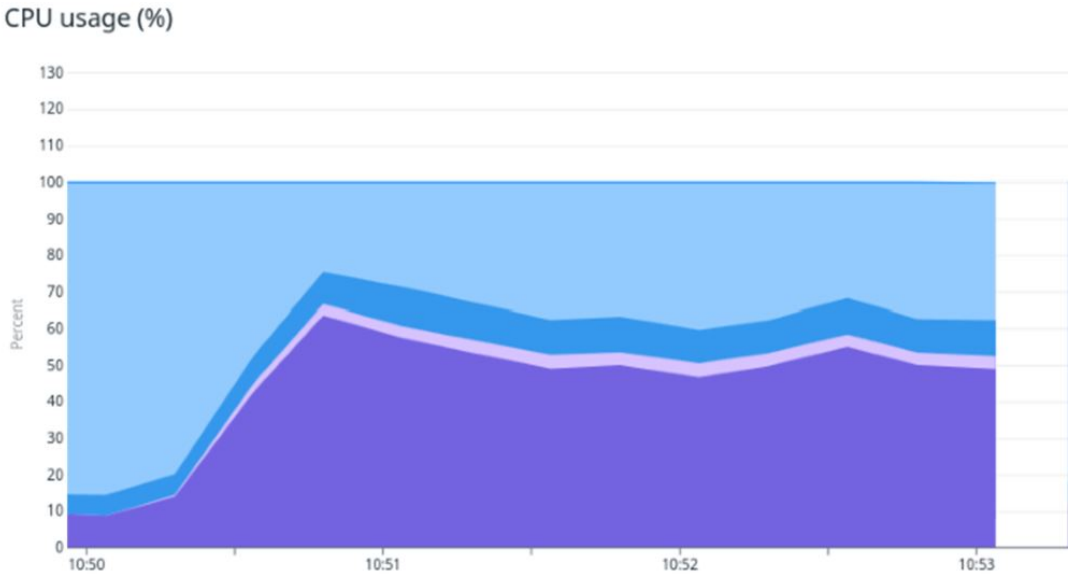


FIGURE 6.16: CPU Usage of Proactive Engine

In the figure ( 6.16), it can be observed that within less than a minute, the entire system was up and running. The maximum CPU usage recorded was 82.17%. The figure also demonstrates that the CPU usage remained relatively stable throughout the execution. The system continued to run for approximately 3 minutes before the Datadog agent was stopped, and the results were obtained. In Figure ( 6.16), an overview of Proactive Engine is provided, depicting three layers. Starting from the top, the light blue layer represents the CPU usage of the system. The middle layer represents the CPU usage for I/O wait. Finally, the dark blue layer represents the CPU usage of the user. However, in Figure ( 6.16), determining the precise process that consumes a certain percentage of the user's CPU usage can be challenging due to the representation of usage in a single color.

To measure the CPU usage of the Navigation 2, the following steps were taken:

1. The Datadog agent was launched at 12:25 minutes.

2. A waiting period of 5 minutes was observed to ensure system stability.

3. At 12:30, the launch file for Navigation2 was executed. This launch file included the launching and connection of Gazebo and RViz. Once these components were successfully connected, it became possible to select the goal for the robot.

In the figure (6.17), it is notable that although the launch file was executed at 12:30, it took nearly 2 minutes for the entire system to fully initialize and become operational. The maximum recorded CPU usage for Navigation 2 was 75.76%. The figure also illustrates the fluctuation in CPU usage immediately after the system was launched, with a subsequent stabilization. Same as the Proactive Engine, the Navigation 2 system also continued to run for 3 minutes before stopping the Datadog agent and obtaining the results.
In figure (6.17), an overview of the Navigation 2 system is presented, showcasing two layers. The light blue layer represents the CPU usage of the system, while the dark blue layer represents the CPU usage of the user. As mentioned earlier, it remains challenging to determine which specific processes account for various percentages of CPU usage.
In Figures (6.18) and (6.19), a detailed analysis is presented, illustrating the

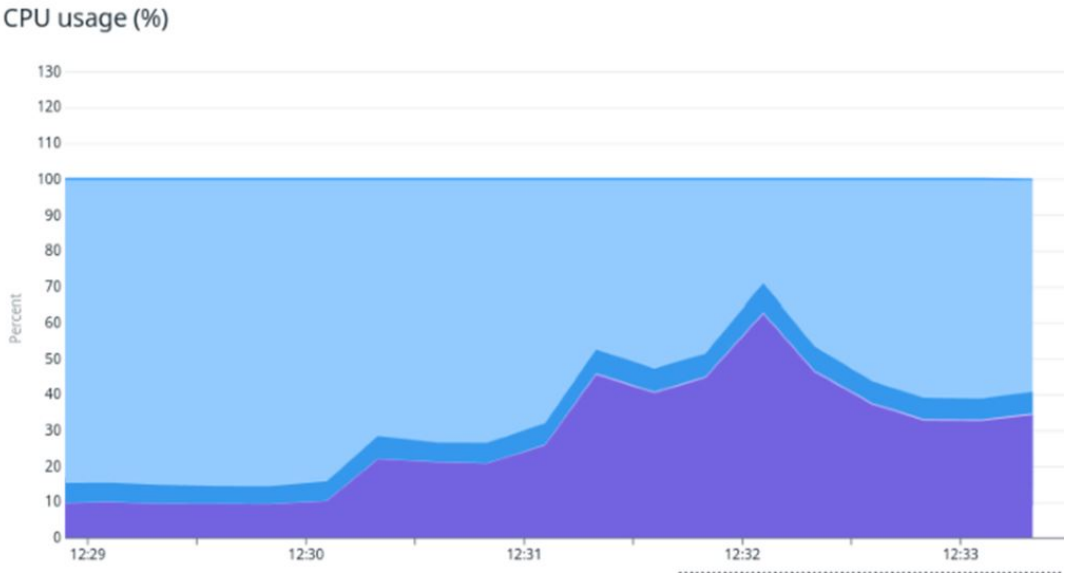


FIGURE 6.17: CPU Usage of navigation 2

breakdown of CPU usage by process in Figure ( 6.16) and Figure (6.17) respectively. These figures offer a comprehensive overview of the percentage of CPU usage allocated to each process, making it easier to identify the specific processes responsible for the user's CPU usage.

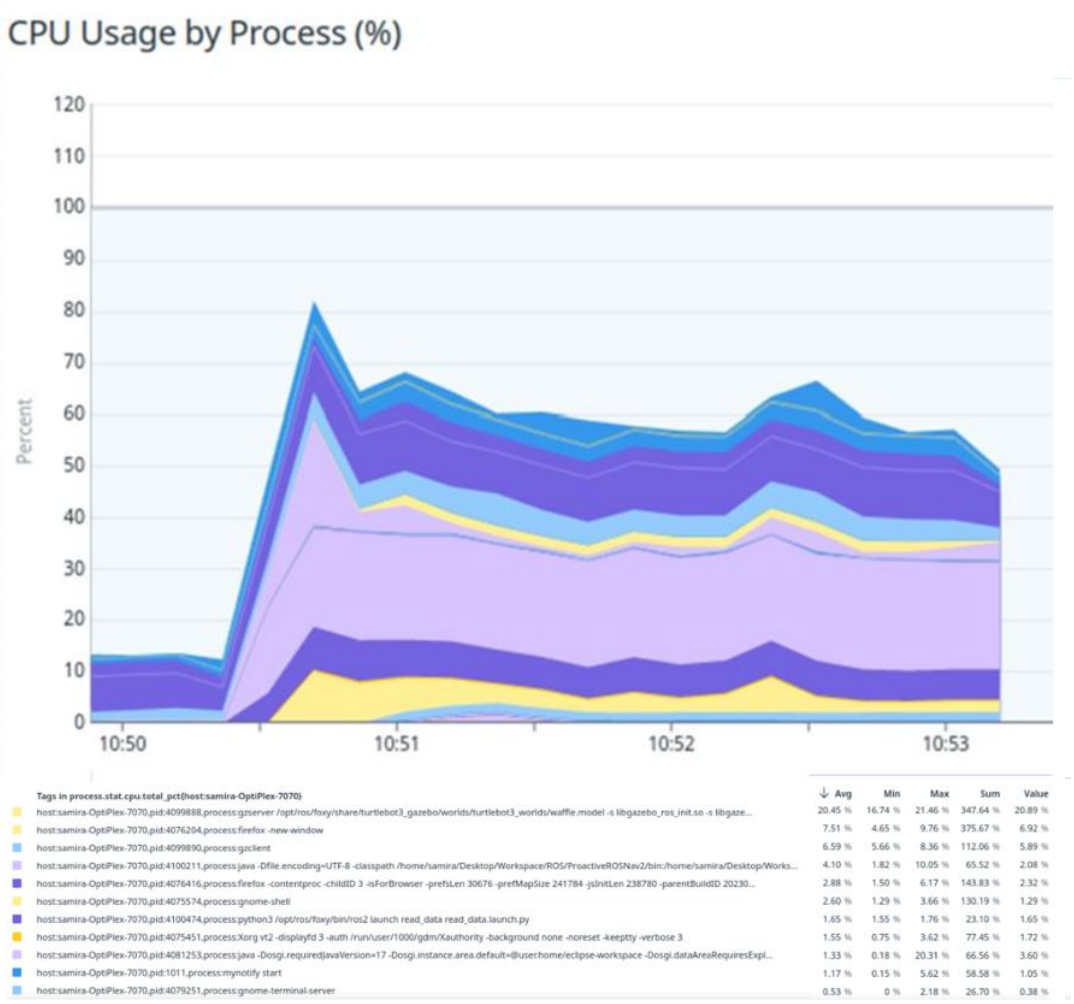


FIGURE 6.18: CPU Usage by process of Proactive Engine

Figure (6.18) presents a comprehensive breakdown of CPU usage in the proactive engine. By examining the figure from the top, we can identify the main processes responsible for consuming CPU resources. These processes are "My notify start", "Xorg", "Firefox", "Firefox", "Gnome-terminal-server", "Gserver", "Gzclient", "Java", and "Python3". These processes play important roles in determining the overall CPU usage of the proactive engine. "My notify start" is likely a custom notification system, "Xorg" represents the X Window System server, "Firefox" is the web browser for monitoring data from Datagod, "Gnome-terminal-server" is the terminal emulator for launching gazebo and ROS nodes, "Gserver" and "Gzclient" are involved in the gazebo simulation, and "Java" is used for running the proactive engine.

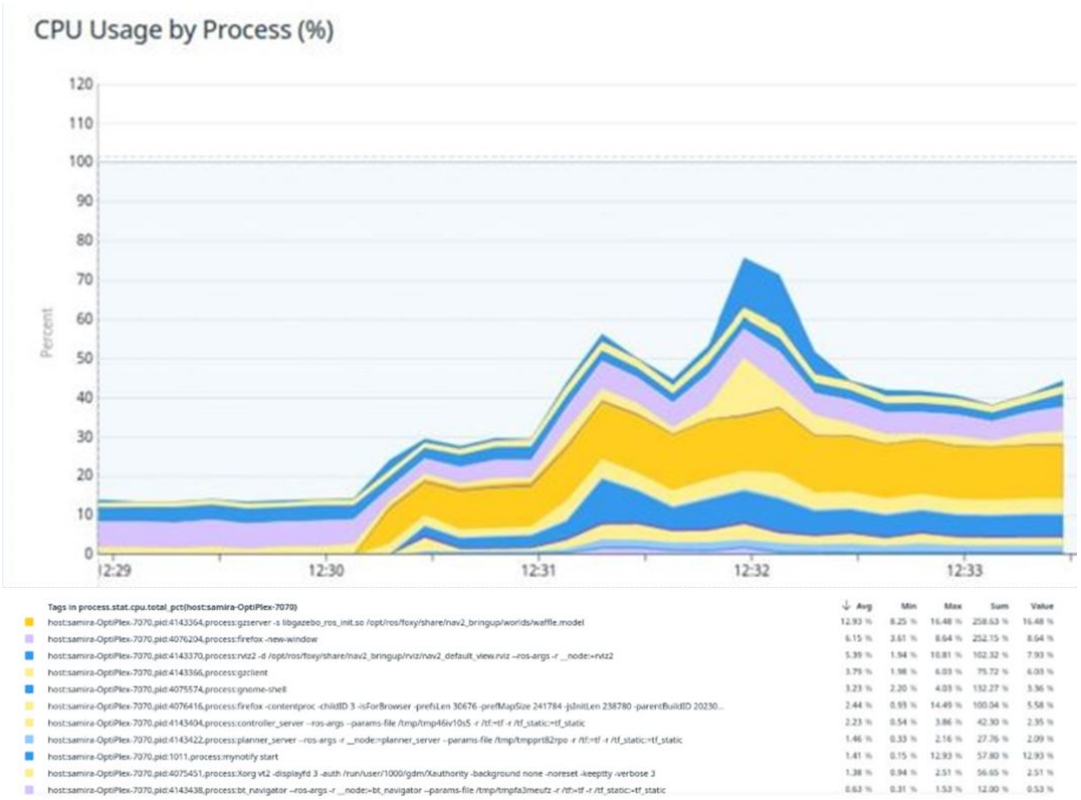


FIGURE 6.19: CPU Usage by Process of Navigation 2

Figure (6.19) shows a detailed breakdown of CPU usage in the Navigation 2 system. The main processes responsible for CPU consumption, from top to bottom, are "My notify start", "Xorg", "Firefox" (purple), "Firefox", "Gzserver", "Gzclient", "Rviz2", "Controller_server", "Planner_Server", and "Bt_navigation". These processes have significant roles in determining the overall CPU usage of the proactive engine. "My notify start" is a custom notification system, "Xorg" is the X Window System server, and "Firefox" is used for data monitoring from Datagod. "Gserver" and "Gzclient" contribute to the

gazebo simulation, while "Rviz2" handles visualization. "Controller_server," "Planner_Server," and "Bt_navigation" are integral components of the navigation system, each serving specific functions in controlling, planning, and implementing behavior trees.

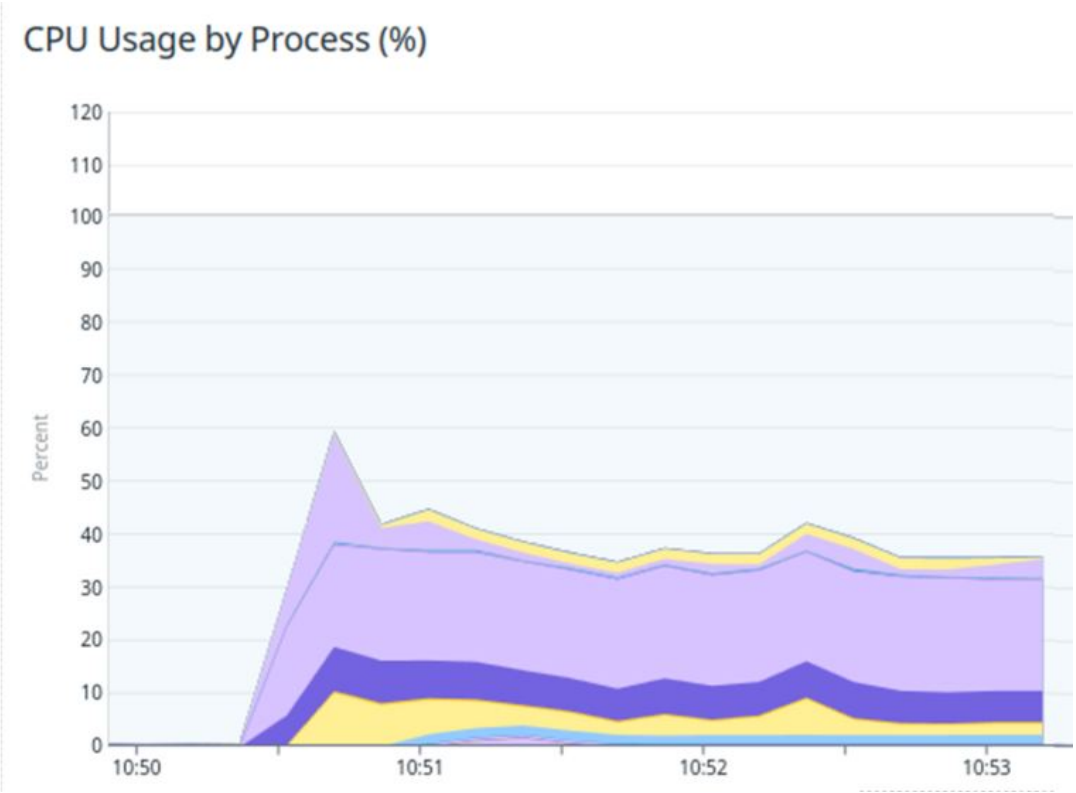


FIGURE 6.20: CPU Usage Breakdown for Proactive Engine Processes

To focus solely on the CPU usage of the Proactive Engine and Navigation 2, we filtered out unrelated processes in Figure (6.20) for the proactive engine and Figure (6.21) for Navigation 2. These figures provide a more detailed view of how each system utilizes CPU resources, displaying the percentage of CPU usage for better analysis. In Figure (6.20), we observe the presence of "Gserver," "Gzclient," "Java," and "Python3" processes exclusively.

In Figure (6.21), we can see exclusively "Gzserver," "Gzclient," "Rviz2," "Controller_server," "Planner_Server," and "Bt_navigation" processes.

### 6.3.2 Memory Usage

Monitoring memory usage is crucial for maintaining optimal performance. High levels of memory utilization can result in decreased performance, while

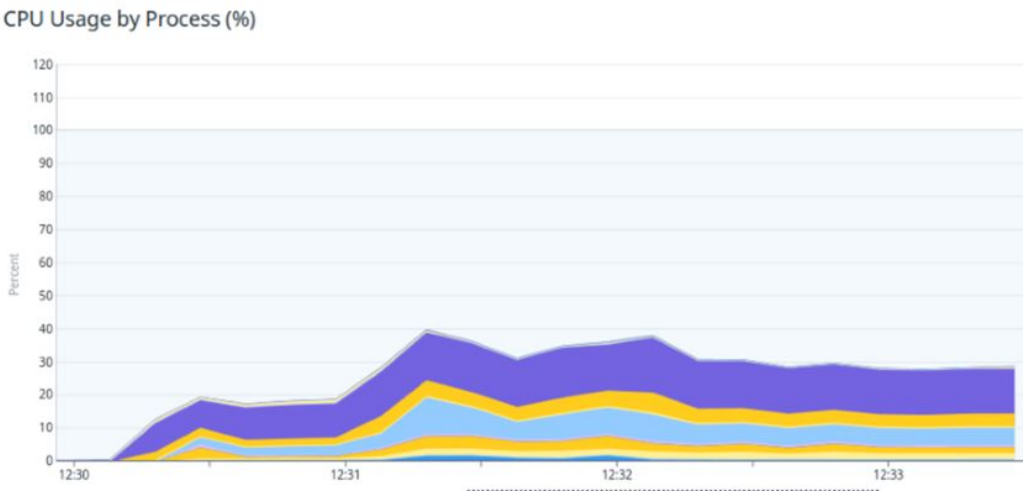


FIGURE 6.21: CPU Usage Breakdown for Navigation 2 Processes

a gradual increase over time may indicate a memory leak. To track and manage memory usage, we rely on datadog, a monitoring software, which helps us identify fluctuations and analyze their causes. By regularly monitoring memory utilization, we can ensure efficient resource allocation and address any issues that may arise.Below, you will find the memory usage statistics for the Proactive Engine and Navigation 2. In Figure (6.22), the memory usage of the proactive engine by process is depicted. Each rectangular shape represents the percentage of memory occupied by a specific process. Starting from the highest to the lowest, we observe the following utilization: 65.13% of the memory is free, 6.15% is allocated to java, 4.66% to firefox, 3.11% to firefox, 2.47% to mysql, 2.19% to gnome-shell, 2.13% to gzserver, 1.8% to gzclient, 1.64% to WebKitWebProcess, 1.58% to mynotify, 1.49% to Web Content, 1.29% to snap-store, 1.2% to jetbrains-toolbox, 1.09% to agent, 0.71% to process-agent, 0.67% to WebExtensions, 0.65% to postgres, 0.64% to Xorg, 0.56% to ros2, 0.43% to python3, and 0.41% to CTERAAgent.

In Figure (6.23), the memory usage of Navigation 2 by process is visualized. Each rectangular shape represents the percentage of memory occupied by a specific process. The utilization, listed from highest to lowest, is as follows: 72.9% of the memory is free, 3.69% for firefox, 2.9% for firefox, 2.47% for mysql, 2.18% for gnome-shell, 2% for gzserver, 1.74% for gzclient, 1.58% for mynotify, 1.49% for Web Content, 1.37% for jetbrains-toolbox, 1.29% for snap-store, 1.16% for rviz2, 1.07% for agent, 0.67% for WebExtensions, 0.66% for Xorg, 0.65% for postgres, 0.58% for process-agent, 0.43% for python3, 0.41%

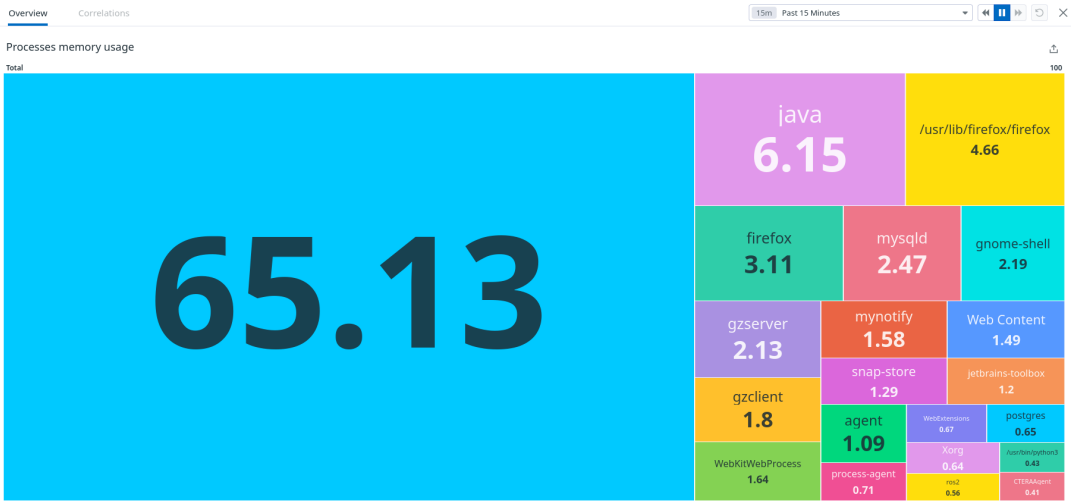


FIGURE 6.22: CPU Usage of Proactive Engine

for CTERAAgent, 0.39% for gnome-terminal-server, and 0.37% for evolution-alarm-notify.

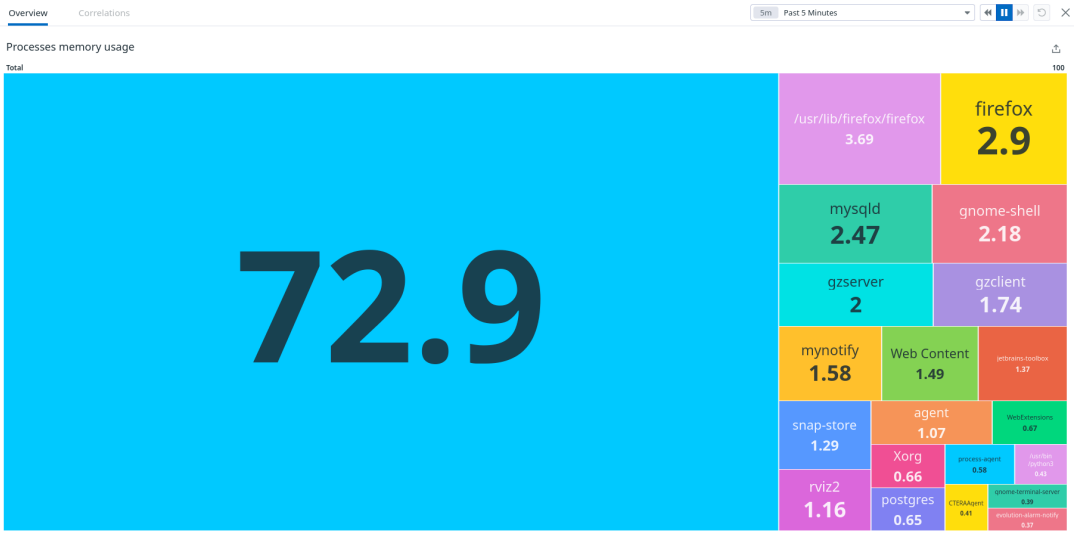


FIGURE 6.23: CPU Usage of Navigation 2

### 6.3.3 Summary

In summary, the Proactive Engine quickly became operational in under a minute, maintaining stable CPU usage throughout execution ( peaking at 82.17% ). The figure for the Proactive Engine displayed three layers representing system and user CPU usage, as well as I/O wait for database operations. However, identifying specific processes contributing to CPU usage was challenging due to the color representation. In contrast, Navigation 2

took around two minutes to initialize, with initial CPU fluctuations before stabilizing. The maximum CPU usage recorded for Navigation 2 was 75.76%. The figure for Navigation 2 showcased system and user CPU usage, without the inclusion of I/O wait, making it similarly challenging to pinpoint specific processes. To analyze the systems in detail, we utilized the breakdown of CPU usage by process, enabling a comprehensive comparison between the two systems. To further enhance clarity, we isolated and presented the core system processes for each system, allowing for a distinct view of the unique processes employed by each system. Upon examination, it became apparent that while both systems shared certain processes, the Proactive Engine specifically employed Java for engine launching and database utilization.

In summary, we conducted an analysis comparing the memory usage statistics of the Proactive Engine and Navigation 2. Figure (6.22) visually represented the memory usage of the Proactive Engine, with each rectangle indicating the percentage of memory occupied by a specific process. Similarly, Figure (6.23) illustrated the memory usage of Navigation 2. While both systems shared many common processes, the Proactive Engine stood out by utilizing additional processes such as Java (6.15%). In summary, an analysis was conducted to compare the memory usage statistics of the Proactive Engine and Navigation 2 systems. Figure (6.22) visually represented the memory usage of the Proactive Engine, with each rectangle indicating the percentage of memory occupied by a specific process. Similarly, Figure (6.23) illustrated the memory usage of Navigation 2. While both systems shared many common processes, the Proactive Engine stood out by utilizing additional processes such as Java (6.15%) and MySQL (3.11%) that were not present in Navigation 2. This variance in process usage accounted for the disparity in free memory between the two systems. The table (6.1) provided a comparison of the memory usage percentages for each process in both systems, showcasing the differences in memory utilization.

## 6.4 Maintaining and Extending Software Systems

In this section, we will compare the features of maintaining and extending systems in Navigation 2 and Proactive Engine.
Let's examine an issue that the core designer of Navigation 2 published on GitHub [Orduno, 2019]. The figures and definitions referenced here are from that reference. According to Jay Wright Forrested, a mental model can be defined as follows: "The image of the world around us, which we carry in

| Process | Proactive Engine | Navigation 2 |
|---|---|---|
| Free Memory | 65.13% | 72.9% |
| Java | 6.15% | - |
| Firefox | 4.66% | 3.69% |
| firefox | 3.11% | 2.9% |
| MySQL | 2.47% | 2.47% |
| Gnome Shell | 2.19% | 2.18% |
| Gzserver | 2.13% | 2% |
| Gzclient | 1.8% | 1.74% |
| WebKitWebProcess | 1.64% | - |
| Mynotify | 1.58% | 1.58% |
| Web Content | 1.49% | 1.49% |
| Snap Store | 1.29% | 1.29% |
| Jetbrains Toolbox | 1.2% | 1.37% |
| Rviz2 | - | 1.16% |
| Agent | 1.09% | 1.07 |
| Process Agent | 0.71% | 0.58% |
| WebExtensions | 0.67% | 0.67% |
| Postgres | 0.65% | 0.65% |
| Xorg | 0.64% | 0.66% |
| ROS2 | 0.56% | - |
| Python3 | 0.43% | 0.43% |
| CTERAAgent | 0.41% | 0.41% |
| Gnome Terminal Server | - | 0.39% |
| Evolution Alarm Notify | - | 0.37% |

TABLE 6.1: Memory Usage Comparison: Proactive Engine vs. Navigation 2

our head, is just a model. Nobody imagines the entire world, government, or country in their head. They only select concepts and the relationships between them to represent the real system." Based on this model, the designer of Navigation 2 developed the navigation model. As shown in Figure (6.24), the world model is populated with information obtained from sensing & perception, and mapping. This information is then supplied to the navigation sub-modules, which include Global Path Planning, Local Path Planning which are Obstacle Avoidance & Control, and Motion Primitives & Recovery. This represents the current world model for Navigation 2. The design team aims to create a world model for 2D navigation with the following objectives:

- Supporting different levels of navigation
- Adding support for various types of planners and controllers

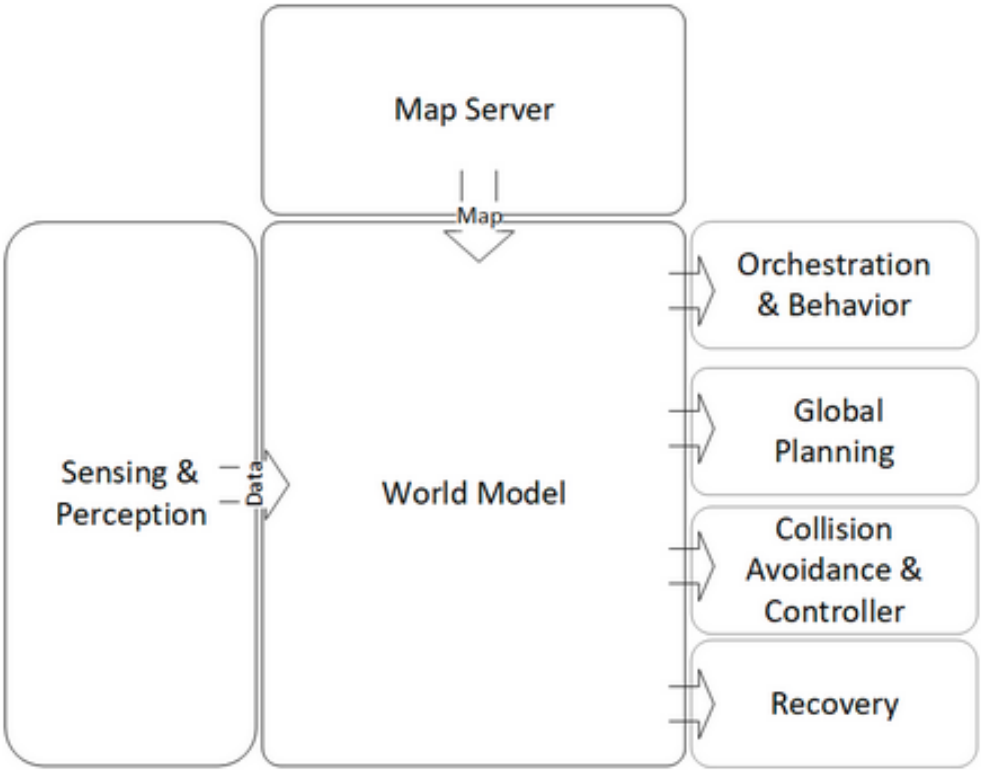


FIGURE 6.24: World Model Navigation 2 [Orduno, 2019]

- Consolidating the world model
- Defining a clean interface
- Avoiding code replication
- Improving the integration of perception pipelines

They proposed several phases. In the first phase, they suggested separating the world model from the clients and making them separate nodes. The second phase involved proposing new modules and porting the current costmap-based world model. The main objective of this phase was to remove the dependency between the core representation and the type of client. In the subsequent phase, they suggested extending this approach by introducing other map formats (beyond grid-based maps) and perception pipelines. Additionally, they proposed supporting multiple internal representations. Ultimately, they found that the new designs not only failed to achieve the objectives but also complicated the system. As a result, they decided to stick with the current design.

In our implementation of the Proactive Engine, we have successfully achieved

a better separation of concerns by treating each objective as a separate scenario. Extending the code does not necessarily lead to increased complexity. We have designed the system to easily incorporate different algorithms for various robot types and platforms without the need for modifying the existing code or additional configurations. This allows for seamless code extension without introducing unnecessary complexity. For example, in ourp implementation, the path computation and robot control tasks have separate scenarios, making it easier to add new algorithms or functionalities to each scenario without affecting the overall system complexity. By focusing on separation of concerns and providing a flexible code base for extension, our implementation offers improved maintainability and adaptability in robotic software systems. It simplifies development and facilitates the evolution of robotic applications to meet different requirements and platforms.
In conclusion, the comparison between Navigation 2 and the Proactive Engine revealed different approaches to maintaining and extending systems. Navigation 2 aimed to create a 2D navigation world model with various objectives, but the new designs did not meet the goals effectively. In contrast, the Proactive Engine implementation emphasized easy code extension without increased complexity. By incorporating separate scenarios for each scenario, our implementation allowed for seamless addition of new algorithms or functionalities, enhancing maintainability and adaptability in robotic software systems.

## 6.5 Dynamic Change of Decision Making

Our implementation incorporates various rules and strategies that can be applied to the robot's behavior dynamically, without the need to relaunch the system. As we discussed in the implementation of the Proactive Engine, the presence of a feedback loop enables the selection of different strategies, resulting in a completely different behavior without requiring any code modifications during runtime. Our implementation enables a high level of flexibility in how the system behaves. By utilizing the proactive engine, we can make real-time adjustments and fine-tune the robot's sterategy and decision-making scenarios. This adaptability allows us to respond to changing conditions and requirements without interrupting or restarting the system. As a result, the system provides a smooth and responsive experience. Overall, our implementation empowers robotic systems with the ability to dynamically

alter their behavior using different rules and strategies at runtime. This flexibility ensures that the system can respond effectively to changing circumstances and perform optimally without the need for code changes or system relaunches.

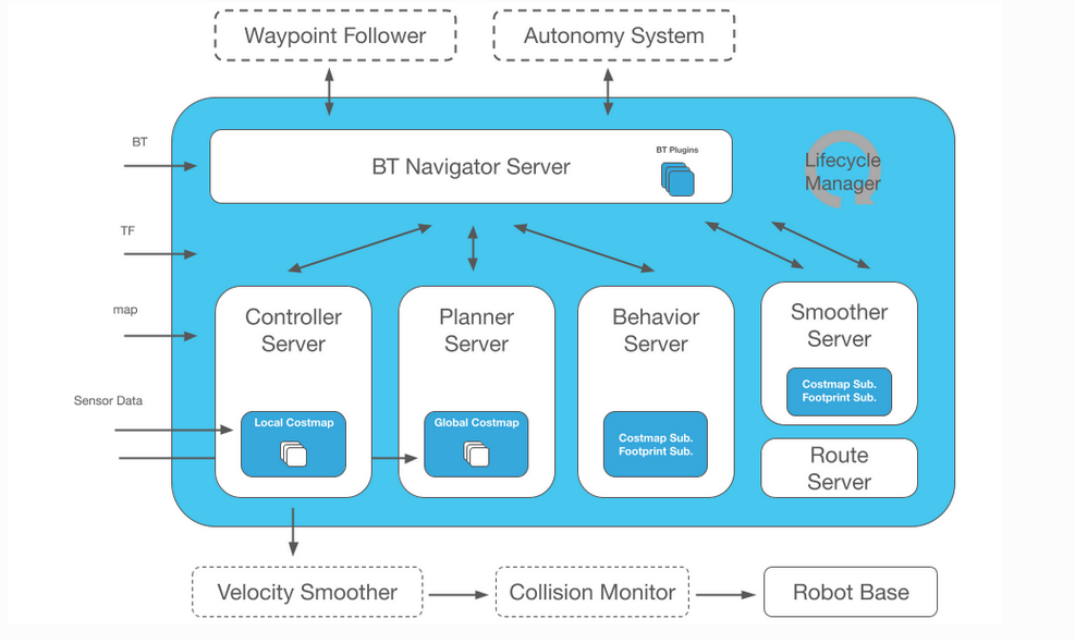


FIGURE 6.25: Navigation 2 Behaviour Tree [Macenski et al., 2020]

In the Navigation 2 implementation, as shown in Figure (6.25), they provide a set of strategy plugins that allow for different strategies or algorithms to be used for planner, controller, and behavior plugins. However, in order to make changes, it is currently required to stop the system, edit the corresponding file, save it, and then relaunch the system[Macenski et al., 2020].
In our implementation, we have introduced a runtime capability that enables us to make these changes without the need to stop and restart the system. This flexibility allows for more efficient development and testing, as adjustments can be made on-the-fly during runtime.
In conclusion, our implementation offered a robust solution that enabled the system to effectively respond to changing circumstances and perform optimally without the need for code changes or system relaunches. The ability to dynamically adjust behavior using different rules and strategies at runtime enhanced the system's adaptability and performance. Unlike Navigation2, which necessitated system interruption and file editing, our implementation allowed for real-time changes during runtime, providing greater flexibility and efficiency.

## 6.6 Conclusion

In conclusion, our comparison between the Proactive Engine and Navigation 2 involved several tools and aspects. We evaluated the systems based on compile time using CodeMR software, run time using Datadog, maintenance and extension capabilities, and the ability to handle dynamic changes in decision-making.

The Proactive Engine and Navigation 2 displayed substantial differences in code size, complexity, coupling, and lack of cohesion during compile time. The Proactive Engineexhibited a notably smaller codebase with lower complexity levels, fewer instances of medium-high complexity, coupling, and lack of cohesion compared to Navigation 2. Additionally, the project size of the Proactive Engine was relatively smaller. The figures and visualizations provided further confirmed these disparities, highlighting variations in package structures, class sizes, and C3 levels between the two projects. Overall, the Proactive Engine demonstrated superior codebase efficiency and effective complexity management compared to Navigation 2.

The runtime analysis indicated that the Proactive Engine and Navigation 2 displayed disparities in system performance and memory usage. The Proactive Engine exhibited faster initialization, maintained stable CPU usage, and utilized additional processes such as Java and MySQL. Furthermore, the memory usage differed, with the Proactive Engine employing more processes compared to Navigation 2.

The comparison between Navigation 2 and the Proactive Engine emphasized different approaches to system maintenance and extension. Navigation 2 aimed to create a 2D navigation world model but faced challenges in meeting its objectives effectively. On the other hand, the Proactive Engine focused on easy code extension without increased complexity, allowing for seamless addition of new algorithms or functionalities. This approach enhanced maintainability and adaptability in robotic software systems.

The Proactive Engine implementation offered a robust solution that allowed the system to respond effectively to changing circumstances without the need for code changes or system relaunches. It provided the ability to dynamically adjust behavior using different rules and strategies at runtime, enhancing adaptability and performance. In contrast, Navigation 2 required system interruption and file editing, limiting real-time changes and overall flexibility.

# Chapter 7

# Conclusion

## 7.1 Revisiting the Research Questions

Let's circle back to each research question and examine the outcomes of our efforts.

**1. How to improve separation of concerns in robotic software engineering? How can software metrics be used to measure the enhancement?**
This study emphasized the vital importance of enhancing the separation of concerns in robotic software engineering. The thesis centered on addressing the absence of this principle by leveraging the proactive scenario approach of the Proactive Engine. This approach empowered developers to strategically divide concerns, resulting in a software architecture that enhanced modularity, maintainability, extendibility, and reusability. The core of our research was intertwined with the first question, exploring how this approach could concretely lead to advancements.
Integrating proactive scenarios molded a versatile software architecture. This blueprint, with its precise concern separation, fostered continuous evolution. The implications were significant: an efficient path for updates, streamlined bug fixing, and simplified maintenance easily extendable without adding complexity and enhancing code reusability. As the architecture enhanced module comprehension, it enabled seamless extensibility and adaptability. This envisioned outcome was a software system that thrived in the ever-changing software industry.
In this context, the strategic use of software metrics became a powerful tool for quantifying the improvement in the separation of concerns. These metrics provided an objective perspective through which we could assess the effectiveness of our proposed methodology. By analyzing these metrics, we could track the tangible progress towards our overarching goal. In conclusion, this study emphasized the crucial significance of the separation of concerns in

shaping the future of robotic software engineering. Through the adoption of innovative methodologies like the Proactive Engine, we aimed to redefine software architectures and enhance their adaptability. As the software landscape continued to evolve, a modular, easily comprehensible, and responsive system became an invaluable asset.

**2. Is the proactive computing paradigm, implemented through a rule-based proactive engine, an adequate coding approach for addressing our first research question?**

The exploration of the second research question, centered on the adequacy of the proactive computing paradigm in addressing the first research question, has been a journey of comprehensive analysis. We implemented a software model that relied on the Proactive Engine, a rule-based proactive system seamlessly integrating object-oriented principles with rule-based systems. The Proactive Engine framework facilitated the parallel operation of numerous scenarios, ensuring both efficiency and autonomy. This approach, characterized by the independent and decentralized nature of each scenario, eliminated the need for explicit inter-scenario communication, thereby promoting streamlined architecture and scalability.

Our inquiry then transitioned to evaluating the viability of the proactive computing paradigm, particularly through the implementation of a rule-based proactive engine, as a coding approach to tackle the initial research question. This primary question revolved around the enhancement of the separation of concerns in robotic software engineering. Within this context, we carefully scrutinized the conceptualized model in relation to the second research question.

Through rigorous examination and analysis, we have established that the proactive computing paradigm, represented by our rule-based proactive engine, stands as an effective coding approach for addressing the enhancement of separation of concerns. The decentralized and scenario-based nature of the Proactive Engine aligns well with the goal of modularizing concerns within robotic software systems. This paradigm not only offers an effective means of addressing the initial research question but also opens avenues for future advancements in the field of robotic software engineering.

In conclusion, our exploration into the proactive computing paradigm has affirmed its aptness as a coding approach to tackle the challenge of enhancing the separation of concerns. This validation paves the way for refined software architectures that embody modularity and scalability, underscoring the

pivotal role of innovative coding methodologies in shaping the future landscape of robotic software engineering.

**3. How can conflict handling be effectively managed in proactive decision-making scenarios where conflicting recommendations arise from multiple independent objective-based scenarios?**
In conclusion, the examination of the third research question, focused on effectively managing conflict handling in proactive decision-making scenarios where conflicting recommendations originate from multiple independent objective-based scenarios, has yielded valuable insights. One of the pivotal components of our software system, the Decision Making (DM) scenario, has emerged as a central piece in the decision-making process. This DM scenario acts as a pivotal point for data reception, acquiring recommendation commands from both the Controller and Planner modules. Each command carries a designated priority, meticulously assigned during the scenario implementation phase. This deliberate incorporation of priorities in the creation of command recommendations ensures the seamless integration of new scenarios within the DM scenario. Importantly, this integration is achieved without necessitating extensive modifications. This adaptive approach empowers the system to expand and evolve, accommodating new components while minimizing disruption to the decision-making process. This strategic design choice underscores the importance of conflict resolution and decision-making in the context of proactive systems, positioning our software architecture for versatility and longevity in the dynamic landscape of robotic software engineering.

**4. How can smart and self-managing behavior be designed and implemented in the system to address our third research question, enabling dynamic changes in strategy?**
The exploration of the fourth research question, focused on designing and implementing smart and self-managing behavior in the system to address the third research question and enable dynamic changes in strategy, has culminated in significant advancements. The Decision Making (DM) scenario, a key component within our system, has played a pivotal role in realizing these objectives.

The orchestration of recommendation commands within the Decision Making scenario was meticulously guided by their assigned priorities. This orchestration ultimately led to the determination of the optimal decision, subsequently dispatched to the robot for execution. However, the DM scenario's significance extended beyond this foundational function. Notably, the incorporation of a feedback loop within this scenario introduced a profound enhancement. This loop not only facilitated the execution of recommended commands but also extended its influence to the strategy scenario. This synergy fostered real-time adaptation in the system's behavior, guided by the dynamically planned strategy.
Importantly, the architecture ensured the isolation of the DM scenario from other components. Each scenario independently generated command recommendations, which were conveyed indirectly to the DM scenario through the database. This architectural choice ensured the autonomy and encapsulation of decision-making roles, while simultaneously facilitating seamless communication among various components of the system. This architectural integrity underpinned the adaptability of our software framework, positioning it to effectively navigate and respond to dynamic changes.
In conclusion, the investigation of the fourth research question has resulted in the realization of a self-managing and dynamically responsive ecosystem within our software system. The symbiotic relationship between the Decision Making and strategy scenarios, empowered by the feedback loop, showcases the system's ability to autonomously adapt in real-time. This achievement underscores the sophistication of our software architecture, offering a tangible pathway to robust and adaptable robotic software engineering solutions.

## 7.2 Future work

Drawing from our previous discussions, the future development of our robotic software system holds promising opportunities for innovation and advancement. We see two key directions for our future work. First, we aim to complete the implementation of the recovery scenarios within the system. Second, we plan to seamlessly integrate machine learning techniques to enhance decision-making. These avenues promise to significantly enhance the system's capabilities and strengthen its resilience.

### 7.2.1 Recovery Scenario Implementation Completion

In our future work, a critical path involves integrating a recovery scenario into our system. Recoveries are essential in fault-tolerant systems, designed to autonomously handle unknown or failure conditions. For instance, if the perception system encounters faults resulting in a cluttered environmental representation with fake obstacles, the clear recovery can trigger to allow the robot to move. Likewise, if the robot gets stuck due to dynamic obstacles or control issues, actions like backing up or spinning in place, if feasible, can help it maneuver to a better position. In extreme cases of total failure, a recovery may notify an operator for assistance.

Recovery scenarios can be complex, but our implementation approach breaks them down into manageable sections. Our system's design inherently supports the addition of new scenarios. The modular architecture we've meticulously crafted, with a clear separation of concerns, simplifies the integration of a recovery scenario. This sets us apart from systems like Navigation 2, which grapple with complexities when extending recovery features due to their intricate structures.

Our system's design philosophy ensures that extending the recovery scenario will be a seamless process. While Navigation 2 faces challenges in integrating new features into recovery scenarios, our system's modular nature grants a unique advantage. This modularity allows us to integrate additional functionalities effortlessly without jeopardizing the system's stability or introducing unnecessary complexities.

Incorporating a recovery scenario can significantly enhance our system's ability to respond to unexpected situations and errors. Moreover, the ease of extending this recovery scenario reflects our commitment to building a system that is both effective and highly adaptable. Leveraging our existing architecture, we can confidently explore new functionalities and features without the constraints experienced by other systems.

Through strategic integration and a modular design, we are poised to revolutionize the field of robotic software engineering, contributing to the development of intelligent and resilient robotic systems. This forward-looking approach underscores our dedication to shaping a future where adaptability and innovation drive the forefront of robotics technology.

### 7.2.2 Machine Learning for Enhanced Decision-Making

While our current research has shed light on promising avenues for addressing challenges in separation of concerns, proactive computing, and dynamic decision-making within robotic software engineering, there is still much fertile ground for future exploration. One particularly noteworthy path involves the integration of machine learning techniques into the decision making process, paving the way for even more intelligent and adaptable robotic systems.

Machine learning, renowned for its ability to identify patterns, learns from data, and makes predictions, offers a compelling opportunity to enhance decision making capabilities within our software architecture. By incorporating machine learning algorithms into the Decision Making (DM) scenario, we can empower the system to learn from past experiences, adapt to changing contexts, and optimize decision outcomes.

In the realm of machine learning, we can utilize data obtained from the Proactive Engine's input and output as a training dataset. This dataset will undergo the training process, and once we have a well-trained neural network, it can be seamlessly integrated as an additional recommendation system into the decision-making scenario.

Furthermore, the incorporation of machine learning can enable predictive analytics, allowing the system to anticipate potential conflicts or scenarios based on historical data. This predictive capability could guide the decision-making process, proactively averting issues before they manifest, thereby enhancing the system's overall efficiency and reliability.

While the integration of machine learning into decision-making poses challenges such as data collection, training, and model integration, the potential benefits are substantial. Not only could it elevate decision-making quality, but it could also lead to the creation of truly autonomous and adaptive robotic systems, continuously enhancing their performance based on real-world experiences. This represents an exciting frontier in the evolution of robotic software engineering.

In conclusion, the integration of machine learning techniques into decision-making processes stands as a promising direction for future research. By augmenting our software architecture with smarter decision-making capabilities, we can further enhance the adaptability, intelligence, and overall performance of robotic systems in dynamic and unpredictable environments.

# Bibliography

Abdullah, Farooq (2017). "Evaluating Impact of Design Patterns on Software Maintainability and Performance". MA thesis.

Andrade, Luis et al. (2002). "Separating computation, coordination and configuration". In: *Journal of software maintenance and evolution: research and practice* 14.5, pp. 353–369.

Arisholm, Erik, Lionel C Briand, and Audun Foyen (2004). "Dynamic coupling measurement for object-oriented software". In: *IEEE Transactions on software engineering* 30.8, pp. 491–506.

Brooks, Rodney A (1991). "Intelligence without representation". In: *Artificial intelligence* 47.1-3, pp. 139–159.

Cai, Yuanfang et al. (2014). "A decision-support system approach to economics-driven modularity evaluation". In: *Economics-Driven Software Architecture*. Elsevier, pp. 105–128.

Chaychi, Samira, Denis Zampunieris, and Sandro Reis (2023). "Software Model for Robot Programming and Example of Implementation for Navigation System". In: *2023 9th International Conference on Automation, Robotics and Applications (ICARA)*, pp. 75–79. DOI: 10.1109/ICARA56516.2023.10125856.

Dattalo, Amanda (2018). *ROS Introduction*. URL: https://www.ros.org.

Dias, Sergio Marques, Sandro Reis, and Denis Zampunieris (2012). "Personalized, Adaptive and Intelligent Support for Online Assignments Based on Proactive Computing". In: *2012 IEEE 12th International Conference on Advanced Learning Technologies*, pp. 668–669. DOI: 10.1109/ICALT.2012.223.

Dijkstra, Edsger W. (2001). "Go To Statement Considered Harmful". In: *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions*. Ed. by Manfred Broy and Ernst Denert. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 297–300. ISBN: 978-3-642-48354-7. DOI: 10.1007/978-3-642-48354-7_12. URL: https://doi.org/10.1007/978-3-642-48354-7_12.

Dobrican, Remus A. and Denis Zampunieris (2016). "A Proactive Solution, using Wearable and Mobile Applications, for Closing the Gap between the Rehabilitation Team and Cardiac Patients". In: *2016 IEEE International*

*Conference on Healthcare Informatics (ICHI)*, pp. 146–155. DOI: 10.1109/ICHI.2016.23.

Dobrican, Remus-Alexandru, Sandro Reis, and Denis Zampunieris (2013). "Empirical Investigations on Community Building and Collaborative Work inside a LMS using Proactive Computing". In: URL: http://hdl.handle.net/10993/13859.

Ehrhardt, Christian (2010). "CPU time accounting". In: *IBM*. URL: https://en.wikipedia.org/wiki/CPUtime.

Elrad, Tzilla, Robert E Filman, and Atef Bader (2001). "Aspect-oriented programming: Introduction". In: *Communications of the ACM* 44.10, pp. 29–32.

Frantz, Alexandre and Denis Zampunieris (2020). "Separation of Concerns Within Robotic Systems Through Proactive Computing". In: *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*. IEEE, pp. 197–201.

Heckel, Reiko and Gregor Engels (2002). "Relating functional requirements and software architecture: Separation and consistency of concerns". In: *Journal of Software Maintenance and Evolution: Research and Practice* 14.5, pp. 371–388.

Henderson-Sellers, Brian (1995). *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc.

Kumar Chhabra, Jitender and Varun Gupta (2010). "A survey of dynamic software metrics". In: *Journal of computer science and technology* 25, pp. 1016–1029.

Lee, Ming-Chang (2014). "Software quality factors and software quality metrics to enhance software quality assurance". In: *British Journal of Applied Science & Technology* 4.21, pp. 3069–3095.

M, Quigley and et al (2009). "ROS: an open-source Robot Operating System". In: vol. 3. 3.2. URL: https://www.ros.org.

Macenski, S. et al. (2020). *The Marathon 2: A Navigation System*. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).

Macenski, Steven et al. (2022a). "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66, eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

— (2022b). "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66. DOI: 10.1126/scirobotics.abm6074. URL: https://doi.org/10.1126%2Fscirobotics.abm6074.

Neyens, Gilles (2019). "CONFIDENCE-BASED DECISION-MAKING SUPPORT FOR MULTI-SENSOR SYSTEMS". In: p. 106. URL: http://hdl.handle.net/10993/41506.

Nguyen, Vu et al. (2007). "A SLOC counting standard". In: *Cocomo ii forum*. Vol. 2007. Citeseer, pp. 1–16.

Orduno, Carlos A. (2019). *Design the World Model*. URL: https://github.com/ros-planning/navigation2/issues/565.

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

Shirnin, Denis (2014). "Formalising the twofold structure of a proactive system: Proof of concept on deterministic and probabilistic levels". In: URL: http://hdl.handle.net/10993/18929.

Shirnin, Denis, Sandro Reis, and Denis Zampunieris (2013). "Experimentation of proactive computing in context aware systems: Case study of human-computer interactions in e-learning environment". In: *2013 IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support (CogSIMA)*, pp. 269–276. DOI: 10.1109/CogSIMA.2013.6523857.

Shirnin, Denis, Denis Zampunieris, and Sandro Reis (2012). "Design of Proactive Scenarios and Rules for Enhanced e-Learning". In: pp. 253 –258. URL: http://hdl.handle.net/10993/2663.

Tennenhouse, D. (2000). "Proactive computing," in: *Communications of the ACM* 43, pp. 43–50. URL: 43.

Zampunieris, Denis (2006a). "Implementation of a Proactive Learning Management System". In: pp. 3145–3151. URL: http://hdl.handle.net/10993/13857.

Zampunieris, Denis. (2006b). "Implementation of efficient proactive computing using lazy evaluation in a learning management system". In: *Proceedings of" m-ICTE-International Conference on Multimedia and Information & Communication Technologies in Education"*, pp. 886–890.